Name: _____ Term # ____

Homework 1 SOLUTIONS

(400 points)

NOTE: **Chapter 1 of the textbook shows a top down approach of the basis and principles of Computer Graphics. Computer Graphics is also a language that requires a vocabulary of terms. Writing the answers to part A of this Homework commits the vocabulary to muscle memory. Thus, when we will cover the details of the basis and principles of Computer Graphics the vocabulary will already be part of our Computer Graphics world.**

Part B shows you a simple openGL application and allows you to create two objects according to a given specification.

Please note that this Homework is worth 400 points.

A. (300 pts) Paper and Pencil

(Guidelines: *Read the material from the textbook chapter, you can use textbook figures (BB* **RESOURCES › TEXTBOOK POWER POINTS FIGURES***) to exemplify your answer, use keywords, summarize your answer, but the answer* ***cannot be longer the 7 lines!****)*

A computer graphics system is a computer system; as such, it must have all the components of a general-purpose computer system. Let us start with the high-level view of a graphics system, as shown in the block diagram in Figure 1.1. There are five major elements in our system:

1. Input devices
2. Processor
3. Memory
4. Frame buffer
5. Output devices

This model is general enough to include workstations and personal computers, interactive game systems, and sophisticated image-generation systems. Although all the components, with the possible exception of the frame buffer, are present in a standard computer, it is the way each element is specialized for computer graphics that characterizes this diagram as a portrait of a graphics system.

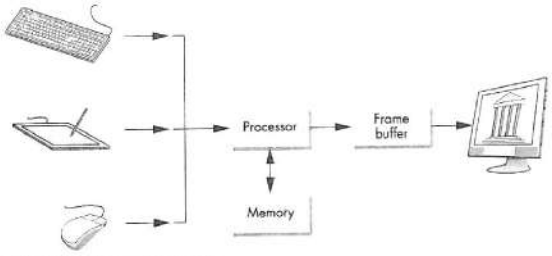


FIGURE 1.1 A graphics system.

1.2.1 Pixels and Frame Buffer

a. Define the frame buffer.

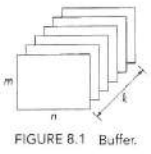


FIGURE 8.1 Buffer.

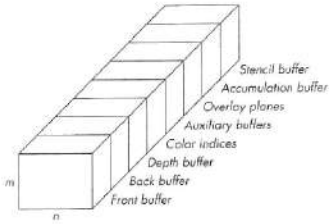


FIGURE 8.2 OpenGL frame buffer.

Overhead photo of a Sun TGX Framebuffer.

Graphics accelerators

As the demand for better graphics increased, hardware manufacturers created a way to decrease the amount of CPU time required to fill the framebuffer. This is commonly called a "graphics accelerator" in the Unix world.

Common graphics drawing commands (many of them geometric) are sent to the graphics accelerator in their raw form. The accelerator then rasterizes the results of the command to the framebuffer. This method can save from thousands to millions of CPU cycles per command, as the CPU is freed to do other work.

While early accelerators focused on improving the performance of 2D GUI systems, most modern accelerators focus on producing 3D imagery in real time. A common design is to send commands to the graphics accelerator using a library such as OpenGL. The OpenGL driver then translates those commands to instructions for the accelerator's graphics processing unit (GPU). The GPU uses those microinstructions to compute the rasterized results. Those results are bit blitted to the framebuffer. The framebuffer's signal is then produced in combination with built-in video overlay devices (usually used to produce the mouse cursor without modifying the framebuffer's data) and any analog special effects that are produced by modifying the output signal. An example of such analog modification was the anti-aliasing technique used by the 3dfx Voodoo cards. These cards add a slight blur to output signal that makes aliasing of the rasterized graphics much less obvious.

Popular manufacturers of 3D graphics accelerators are Nvidia and ATI Technologies.

ANSWER:

(Angel pp. 5-6) ***data structure***

A **framebuffer** is a video output device that drives a video display from a memory buffer containing a complete frame of data. The information in the memory buffer typically consists of color values for every pixel (point that can be displayed) on the screen. Color values are commonly stored in 1-bit monochrome, 4-bit palettized, 8-bit palettized, 16-bit highcolor and 24-bit truecolor formats. An additional alpha channel is sometimes used to retain information about pixel transparency. The total amount of the memory required to drive the frame buffer depends on the resolution of the output signal, and on the color depth and palette size.

> Presently, almost all graphics systems are raster based. A picture is produced as an array—the **raster**—of picture elements, or **pixels**, within the graphics system. As we can see from Figure 1.2, each pixel corresponds to a location, or small area, in

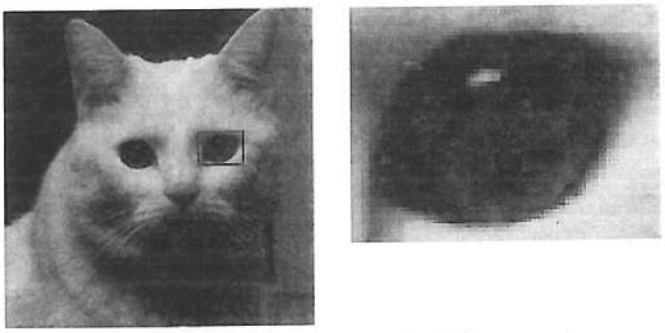


FIGURE 1.2 Pixels. (a) Image of Yeti the cat. (b) Detail of area around one eye showing individual pixels.

the image. Collectively, the pixels are stored in a part of memory called the **frame buffer**. The frame buffer can be viewed as the core element of a graphics system.

The frame buffer usually is implemented with special types of memory chips that enable fast redisplay of the contents of the frame buffer. In software-based systems, such as those used for high-resolution rendering or for generating complex visual effects that cannot be produced in real time, the frame buffer is part of system memory.

b. Define the resolution of the frame buffer.

ANSWER:

(Angel pp. 6)

resolution—the number of pixels in the frame buffer—determines the detail that you can see in the image.

c. Define the depth or precision of the frame buffer.

ANSWER:

(Angel pp. 6-7)

you can see in the image. The **depth**, or **precision**, of the frame buffer, defined as the number of bits that are used for each pixel, determines properties such as how many colors can be represented on a given system. For example, a 1-bit-deep frame buffer allows only two colors, whereas an 8-bit-deep frame buffer allows 2^8 (256) colors. In **full-color** systems, there are 24 (or more) bits per pixel. Such systems can display sufficient colors to represent most images realistically. They are also called **true-color** systems, or **RGB-color** systems, because individual groups of bits in each pixel are

assigned to each of the three primary colors—red, green, and blue—used in most displays.

High dynamic range applications require more than 24-bit fixed point color representations of RGB colors. Some recent frame buffers store RGB values as floating-point numbers in standard IEEE format. Hence, the term *true color* should be interpreted as frame buffers that have sufficient depth to represent colors in terms of RGB values rather than as indices into a limited set of colors.

d. Define the color buffers.

ANSWER:

(Angel pp. 7)

In a very simple system, the frame buffer holds only the colored pixels that are displayed on the screen. In most systems, the frame buffer holds far more information, such as depth information needed for creating images from three-dimensional data. In these systems, the frame buffer comprises multiple buffers, one or more of which are **color buffers** that hold the colored pixels that are displayed. For now, we can use the terms *frame buffer* and *color buffer* synonymously without confusion.

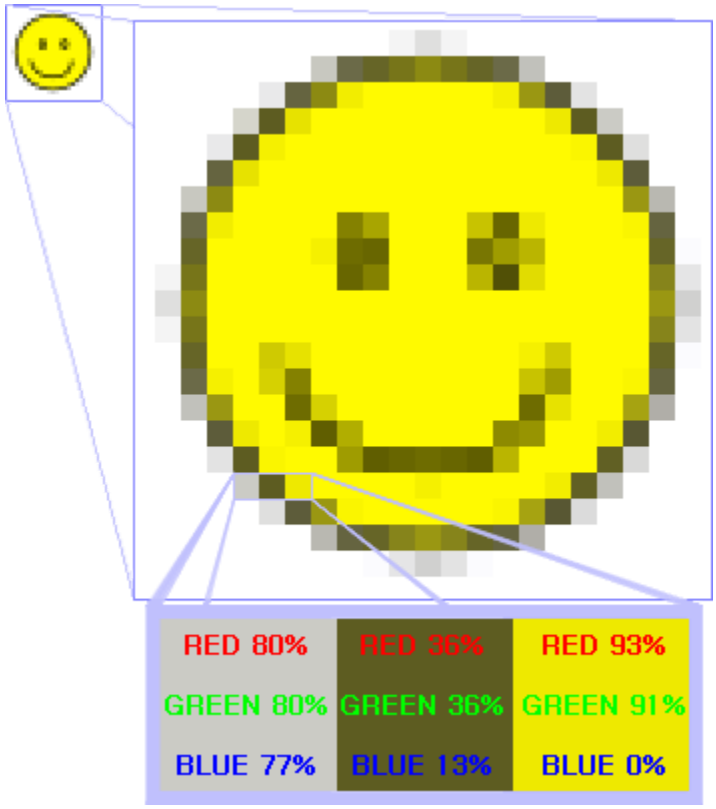
e. Define the GPU.

ANSWER:

(Angel pp. 7)

In a simple system, there may be only one processor, the **central processing unit** (**CPU**) of the system, which must do both the normal processing and the graphical processing. The main graphical function of the processor is to take specifications of graphical primitives (such as lines, circles, and polygons) generated by application programs and to assign values to the pixels in the frame buffer that best represent these entities. For example, a triangle is specified by its three vertices, but to display its outline by the three line segments connecting the vertices, the graphics system must generate a set of pixels that appear as line segments to the viewer. The conversion of geometric entities to pixel colors and locations in the frame buffer is known as **rasterization**, or **scan conversion**. In early graphics systems, the frame buffer was part of the standard memory that could be directly addressed by the CPU. Today, virtually all graphics systems are characterized by special-purpose **graphics processing units** (**GPUs**), custom-tailored to carry out specific graphics functions. The GPU can be either on the motherboard of the system or on a graphics card. The frame buffer is accessed through the graphics processing unit and may be included in the GPU.

1.2.2 Output Devices



The smiley face in the top left corner is a bitmap image. When enlarged, individual pixels appear as squares. Zooming in further, they can be analyzed, with their colors constructed by adding the values for red, green and blue.

a. Define the raster system.

ANSWER:

(Angel pp. 8)

In a raster system, the graphics system takes pixels from the frame buffer and displays them as points on the surface of the display in one of two fundamental ways. In a **noninterlaced** or **progressive** display, the pixels are displayed row by row, or scan line by scan line, at the refresh rate. In an **interlaced** display, odd rows and even rows are refreshed alternately. Interlaced displays are used in commercial television. In an interlaced display operating at 60 Hz, the screen is redrawn in its entirety only 30 times per second, although the visual system is tricked into thinking the refresh rate is 60 Hz rather than 30 Hz. Viewers located near the screen, however, can tell the difference between the interlaced and noninterlaced displays. Noninterlaced displays are becoming more widespread, even though these displays process pixels at twice the rate of interlaced displays.

b. Define the aspect ratio.

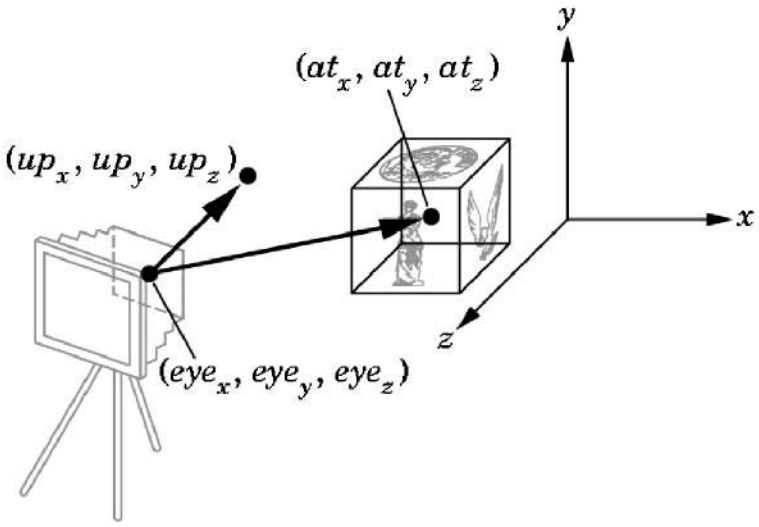
ANSWER:

(Angel pp. 10)

Until recently, most displays had a 4:3 width to height ratio (or **aspect ratio**) that correspondeds to commercial television. In discrete terms, displays started with VGA resolution of 640 × 480 pixels, which was consistent with the number of lines displayed in Standard NTSC video.[2] Computer displays moved up to the popular resolutions of 1024 × 768 (XGA) and 1280 × 1024 (SXGA). The newer High Definition Television (HDTV) standard uses a 16:9 aspect ratio, which is between the older television aspect ratio and that of movies. HDTV monitors display 780 or 1080 lines in either progressive (1080p, 780p) or interlaced (1080i, 780i) modes. Hence, the most popular computer display resolutions are now 1920 × 1080 and 1280 × 720, which have the HDTV aspect ratio, and 1920 × 1024 and 1280 × 768, which have the vertical resolution of XGA and SXGA displays. At the high end, there are now "4K" (4096 × 2160) digital projectors that are suitable for commercial digital movies.

1.3. Images: Physical and Synthetic

1.3.1 OBJECTS AND VIEWERS



a. Define the 3 dimensional objects.

ANSWER:

(Angel pp. 11)

We live in a world of three-dimensional objects. The development of many branches of mathematics, including geometry and trigonometry, was in response to the desire to systematize conceptually simple ideas, such as the measurement of the size of objects and the distance between objects. Often we seek to represent our understanding of such spatial relationships with pictures or images, such as maps, paintings, and photographs. Likewise, the development of many physical devices—including cameras, microscopes, and telescopes—was tied to the desire to visualize spatial relationships among objects. Hence, there always has been a fundamental link between the physics and the mathematics of image formation—one that we can exploit in our development of computer image formation.

Two basic entities must be part of any image-formation process, be it mathematical or physical: *object* and *viewer*. The object exists in space independent of any image-formation process and of any viewer. In computer graphics, where we deal with synthetic objects, we form objects by specifying the positions in space of various geometric primitives, such as points, lines, and polygons. In most graphics systems, a set of locations in space, or of **vertices**, is sufficient to define, or approximate, most objects. For example, a line can be specified by two vertices; a polygon can be specified by an ordered list of vertices; and a sphere can be specified by two vertices that give its center and any point on its circumference.

b. Define the viewer.

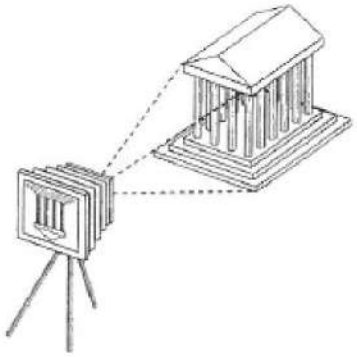


FIGURE 1.7 Camera system.

ANSWER:

(Angel pp. 12)

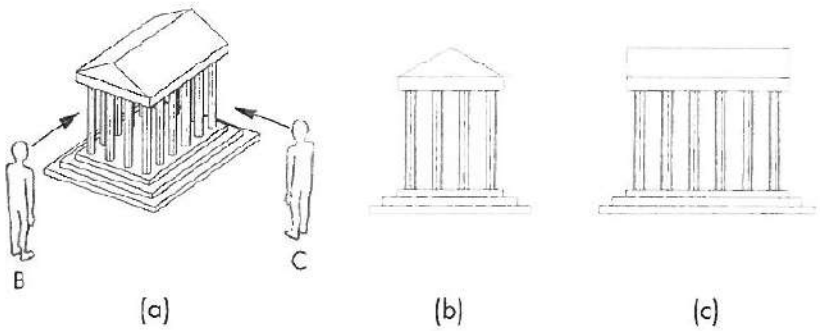


FIGURE 1.6 Image seen by three different viewers. (a) A's view. (b) B's view. (c) C's view.

Every imaging system must provide a means of forming images from objects. To form an image, we must have someone or something that is viewing our objects, be it a person, a camera, or a digitizer. It is the **viewer** that forms the image of our objects. In the human visual system, the image is formed on the back of the eye. In a camera, the image is formed in the film plane. It is easy to confuse images and objects. We usually see an object from our single perspective and forget that other viewers, located in other places, will see the same object differently. Figure 1.6(a) shows two viewers observing the same building. This image is what is seen by an observer A who is far enough away from the building to see both the building and the two other viewers, B and C. From A's perspective, B and C appear as objects, just as the building does. Figures 1.6(b) and (c) show the images seen by B and C, respectively. All three images contain the same building, but the image of the building is different in all three.

Figure 1.7 shows a camera system viewing a building. Here we can observe that both the object and the viewer exist in a three-dimensional world. However, the image that they define—what we find on the film plane—is two dimensional. The process by which the specification of the object is combined with the specification of the viewer to produce a two-dimensional image is the essence of image formation, and we will study it in detail.

1.3.2 LIGHT AND IMAGES

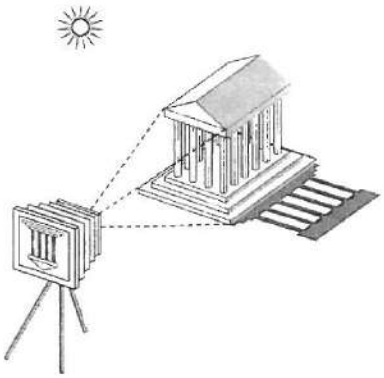


FIGURE 1.8 A camera system with an object and a light source.

a. Define the light.

ANSWER:

(Angel pp. 13)

The preceding description of image formation is far from complete. For example, we have yet to mention light. If there were no light sources, the objects would be dark, and there would be nothing visible in our image. Nor have we indicated how color enters the picture or what the effects of the surface properties of the objects are.

Taking a more physical approach, we can start with the arrangement shown in Figure 1.8, which shows a simple physical imaging system. Again, we see a physical object and a viewer (the camera); now, however, there is a light source in the scene. Light from the source strikes various surfaces of the object, and a portion of the reflected light enters the camera through the lens. The details of the interaction between light and the surfaces of the object determine how much light enters the camera.

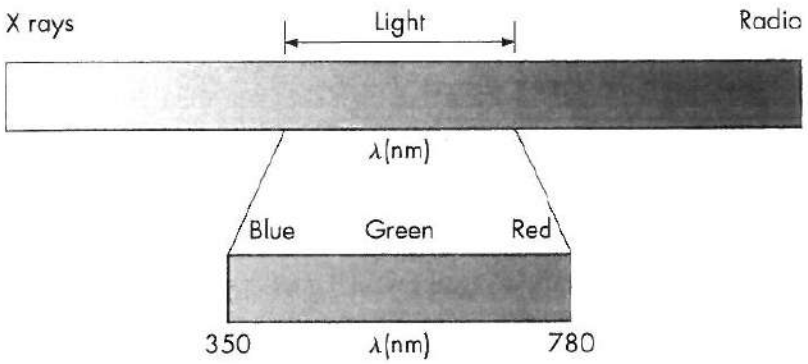


FIGURE 1.9 The electromagnetic spectrum.

Light is a form of electromagnetic radiation. Electromagnetic energy travels as waves that can be characterized by either their wavelengths or their frequencies.[3] The electromagnetic spectrum (Figure 1.9) includes radio waves, infrared (heat), and a portion that causes a response in our visual systems. This **visible spectrum**, which has wavelengths in the range of 350 to 780 nanometers (nm), is called (visible) **light**. A given light source has a color determined by the energy that it emits at various wavelengths. Wavelengths in the middle of the range, around 520 nm, are seen as green; those near 450 nm are seen as blue; and those near 650 nm are seen as red. Just as with a rainbow, light at wavelengths between red and green we see as yellow, and wavelengths shorter than blue generate violet light.

Light sources can emit light either as a set of discrete frequencies or continuously. A laser, for example, emits light at a single frequency, whereas an incandescent lamp emits energy over a range of frequencies. Fortunately, in computer graphics, except for recognizing that distinct frequencies are visible as distinct colors, we rarely need to deal with the physical properties of light, such as its wave nature.

3. The relationship between frequency (f) and wavelength (λ) is $f\lambda = c$, where c is the speed of light.

b. Define the geometric optics.

ANSWER:

(Angel pp. 14)

Instead, we can follow a more traditional path that is correct when we are operating with sufficiently high light levels and at a scale where the wave nature of light is not a significant factor. **Geometric optics** models light sources as emitters of light energy, each of which have a fixed intensity. Modeled geometrically, light travels in straight lines, from the sources to those objects with which it interacts. An ideal **point source** emits energy from a single location at one or more frequencies equally in all directions. More complex sources, such as a light bulb, can be characterized as emitting light over an area and by emitting more light in one direction than another. A particular source is characterized by the intensity of light that it emits at each frequency and by that light's directionality. We consider only point sources for now.

1.3.3 IMAGE FORMATION MODELS

a. Define the "following light from source model".

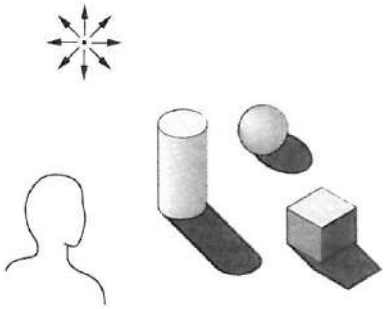


FIGURE 1.10 Scene with a single point light source.

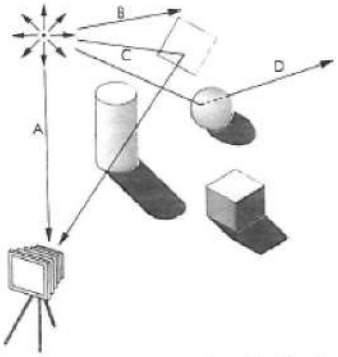


FIGURE 1.11 Ray interactions. Ray A enters camera directly. Ray B goes off to infinity. Ray C is reflected by a mirror. Ray D goes through a transparent sphere.

ANSWER:

(Angel pp. 14)

There are multiple approaches to how we can form images from a set of objects, the light-reflecting properties of these objects, and the properties of the light sources in the scene. In this section, we introduce two physical approaches. Although these approaches are not suitable for the real-time graphics that we ultimately want, they will give us insight into how we can build a useful imaging architecture.

We can start building an imaging model by following light from a source. Consider the scene illustrated in Figure 1.10; it is illuminated by a single point source. We include the viewer in the figure because we are interested in the light that reaches her eye. The viewer can also be a camera, as shown in Figure 1.11. A **ray** is a semi-infinite line that emanates from a point and travels to infinity in a particular direction. Because light travels in straight lines, we can think in terms of rays of light emanating in all directions from our point source. A portion of these infinite rays contributes to the

image on the film plane of our camera. For example, if the source is visible from the camera, some of the rays go directly from the source through the lens of the camera, and strike the film plane. Most rays, however, go off to infinity, neither entering the camera directly nor striking any of the objects. These rays contribute nothing to the image, although they may be seen by some other viewer. The remaining rays strike and illuminate objects. These rays can interact with the objects' surfaces in a variety of ways. For example, if the surface is a mirror, a reflected ray might—depending on the orientation of the surface—enter the lens of the camera and contribute to the image. Other surfaces scatter light in all directions. If the surface is transparent, the light ray from the source can pass through it and may interact with other objects, enter the camera, or travel to infinity without striking another surface.[4] Figure 1.11

Ray tracing and **photon mapping** are image-formation techniques that are based on these ideas and that can form the basis for producing computer-generated images. We can use the ray-tracing idea to simulate physical effects as complex as we wish, as long as we are willing to carry out the requisite computing. Although tracing rays can provide a close approximation to the physical world, it is not well suited for real-time computation.

b. Define the "conservation of energy model".

ANSWER:

(Angel pp. 15)

Other physical approaches to image formation are based on conservation of energy. The most important in computer graphics is **radiosity**. This method works best for surfaces that scatter the incoming light equally in all directions. Even in this case, radiosity requires more computation than can be done in real time.

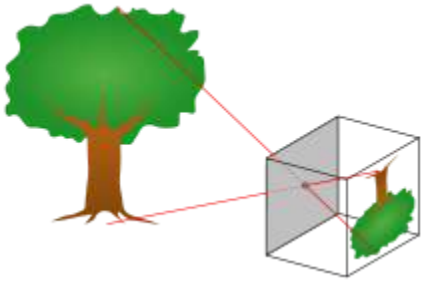
Given the time constraints in image formation by real-time graphics, we usually are satisfied with images that look reasonable rather than images that are physically correct. Such is not the case with images that are non–real-time, such as those generated for a feature film, which can use hours of computer time for each frame. With the increased speed of present hardware, we can get closer to physically correct images in real-time systems

1.4. IMAGING SYSTEMS

1.4.1 The Pinhole Camera.

.A **pinhole camera** is a very simple camera with no lens and a single very small aperture. Simply explained, it is a light-proof box with a small hole in one side. Light from a scene passes through this single point and projects an inverted image on the opposite side of the box. Cameras using small apertures and the human eye in bright light both act like a pinhole camera.

Up to a certain point the smaller the hole, the sharper the image, but the dimmer the projected image. Optimally, the size of the aperture should be 1/100 or less of the distance between it and the screen.



Principle of a pinhole camera. Light rays from an object pass through a small hole to form an image.

The pinhole camera below provides an example of image formation by using a simple geometric model. What is the point (y_p, z_p) , the projection of the point at (y, z). We have eliminated the x coordinate by slicing through the scene and the camera at $x = 0$..

The pinhole camera shown in Figure 1.12 provides an example of image formation that we can understand with a simple geometric model. A **pinhole camera** is a box with a small hole in the center of one side of the box; the film is placed inside the box on the side opposite the pinhole. Initially, the pinhole is covered. It is uncovered for a short time to expose the film. Suppose that we orient our camera along the z-axis, with the pinhole at the origin of our coordinate system. We assume that the hole is so small that only a single ray of light, emanating from a point, can enter it. The film plane is located a distance d from the pinhole. A side view (Figure 1.13) allows us to

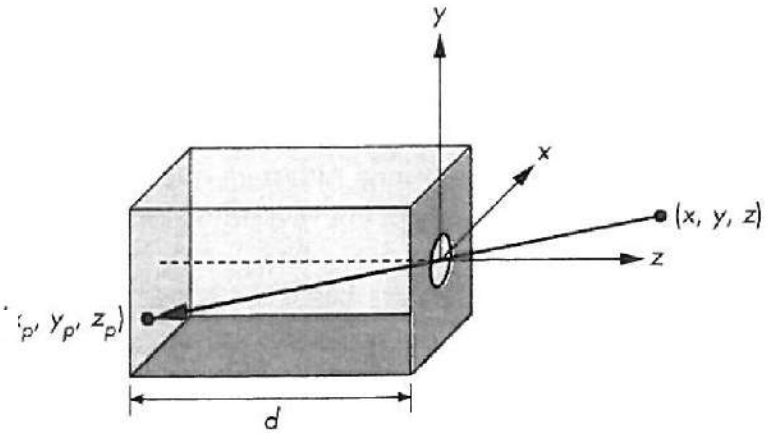


FIGURE 1.12 Pinhole camera.

a. Which variable denotes the depth of the camera?

ANSWER:
(Angel pp. 17)

d

b. What is the point that's being projected?

ANSWER:

(x,y,z)

c. What is the projection of (x,y,z)?

ANSWER:

(x_p, y_p, z_p)

d. What is the value of x in the side view?

ANSWER:

x = 0

e. For the side view in Figure 1.13 show the steps in calculating the value of the projected point y, y_p? (DO NOT COPY THE FORMULA FROM THE BOOK!)

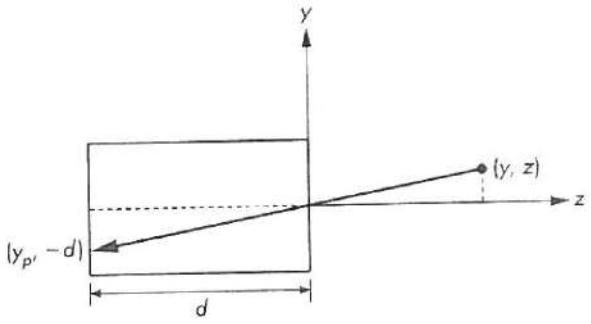
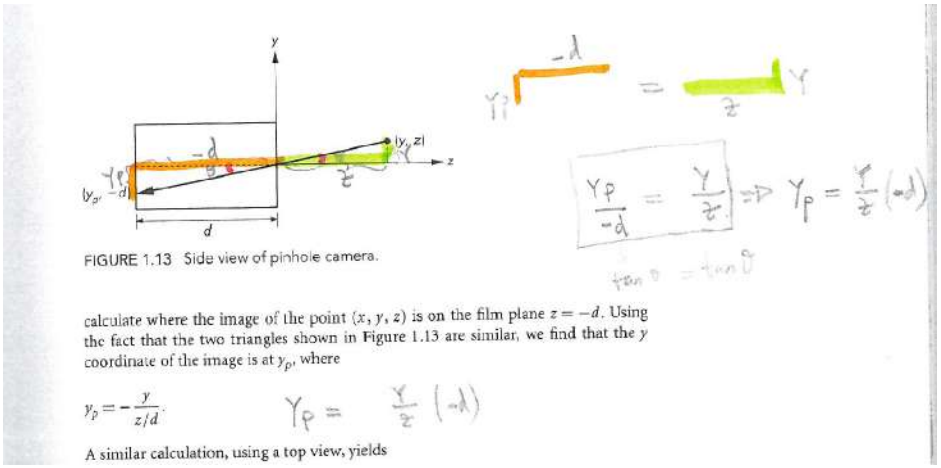


FIGURE 1.13 Side view of pinhole camera.

ANSWER:

(Angel pp. 17)



f. For the side view in Figure 1.14 show the steps in calculating the field of view also known as angle of view θ ? (DO NOT COPY THE FORMULA FROM THE BOOK!)

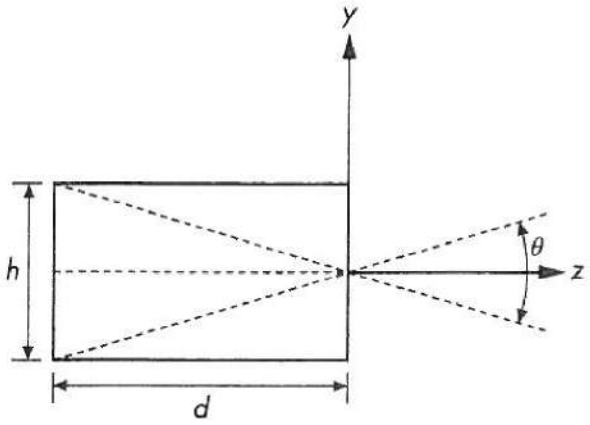
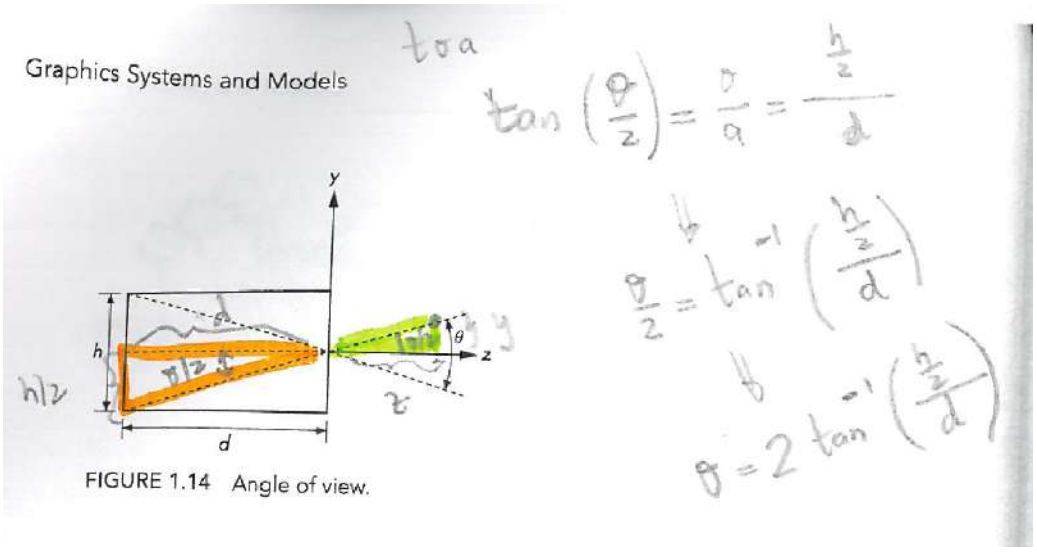


FIGURE 1.14 Angle of view.

The point $(x_p, y_p, -d)$ is called the **projection** of the point (x, y, z). Note that all points along the line between (x, y, z) and $(x_p, y_p, -d)$ project to $(x_p, y_p, -d)$ so that we cannot go backward from a point in the image plane to the point that produced it. In our idealized model, the color on the film plane at this point will be the color of the point (x, y, z). The **field**, or **angle, of view** of our camera is the angle made by the largest object that our camera can image on its film plane. We can calculate the field of view with the aid of Figure 1.14.[5] If h is the height of the camera, then the angle of view θ is

ANSWER:

(Angel pp. 18)



The ideal pinhole camera has an infinite **depth of field**: Every point within its field of view is in focus, regardless of how far it is from the camera. The image of a point is a point. The pinhole camera has two disadvantages. First, because the pinhole is so small—it admits only a single ray from a point source—almost no light enters the camera. Second, the camera cannot be adjusted to have a different angle of view.

The jump to more sophisticated cameras and to other imaging systems that have lenses is direct. By replacing the pinhole with a lens, we solve the two problems of the pinhole camera. First, the lens gathers more light than can pass through the pinhole. The larger the aperture of the lens, the more light the lens can collect. Second, by picking a lens with the proper focal length—a selection equivalent to choosing d for the pinhole camera—we can achieve any desired angle of view (up to 180 degrees). Physical lenses, however, do not have an infinite depth of field: Not all objects in front of the lens are in focus.

For our purposes, in this chapter, we can work with a pinhole camera whose focal length is the distance d from the front of the camera to the film plane. Like the pinhole camera, computer graphics produces images in which all objects are in focus.

1.4.2. The Human Visual System

Our extremely complex visual system has all the components of a physical imaging system, such as a camera or a microscope. The major components of the visual system are shown in Figure 1.15. Light enters the eye through the lens and cornea, a transparent structure that protects the eye. The iris opens and closes to adjust the amount of light entering the eye. The lens forms an image on a two-dimensional structure called the **retina** at the back of the eye. The rods and cones (so named because of their appearance when magnified) are light sensors and are located on the retina. They are excited by electromagnetic energy in the range of 350 to 780 nm.

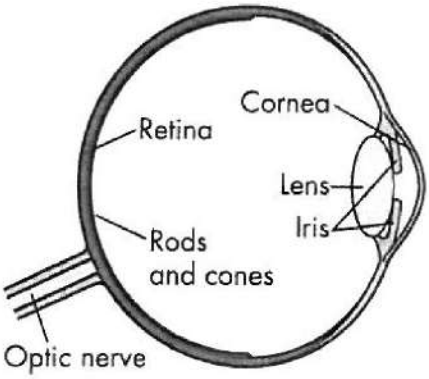


FIGURE 1.15 The human visual system.

a. What is the resolution?

ANSWER:
(Angel pp. 18)

The rods are low-level-light sensors that account for our night vision and are not color sensitive; the cones are responsible for our color vision. The sizes of the rods and cones, coupled with the optical properties of the lens and cornea, determine the **resolution** of our visual systems, or our **visual acuity**. Resolution is a measure of what size objects we can see. More technically, it is a measure of how close we can place two points and still recognize that there are two distinct points.

b. What is the brightness?

ANSWER:
(Angel pp. 18-19)

The sensors in the human eye do not react uniformly to light energy at different wavelengths. There are three types of cones and a single type of rod. Whereas intensity is a physical measure of light energy, **brightness** is a measure of how intense we perceive the light emitted from an object to be. The human visual system does not have the same response to a monochromatic (single-frequency) red light as to a monochromatic green light. If these two lights were to emit the same energy, they would appear to us to have different brightness, because of the unequal response of the cones to red and green light. We are most sensitive to green light, and least sensitive to red and blue.

Brightness is an overall measure of how we react to the intensity of light. Human color-vision capabilities are due to the different sensitivities of the three types of cones. The major consequence of having three types of cones is that instead of having to work with all visible wavelengths individually, we can use three standard primaries to approximate any color that we can perceive. Consequently, most image-production systems, including film and video, work with just three basic, or **primary**, colors. We discuss color in depth in Chapter 2.

The initial processing of light in the human visual system is based on the same principles used by most optical systems. However, the human visual system has a back end much more complex than that of a camera or telescope. The optic nerve is connected to the rods and cones in an extremely complex arrangement that has many of the characteristics of a sophisticated signal processor. The final processing is done in a part of the brain called the visual cortex, where high-level functions, such as object recognition, are carried out. We will omit any discussion of high-level processing; instead, we can think simply in terms of an image that is conveyed from the rods and cones to the brain.

1.5 THE SYNTHETIC-CAMERA MODEL

Bellows Camera

a. Define the projector.

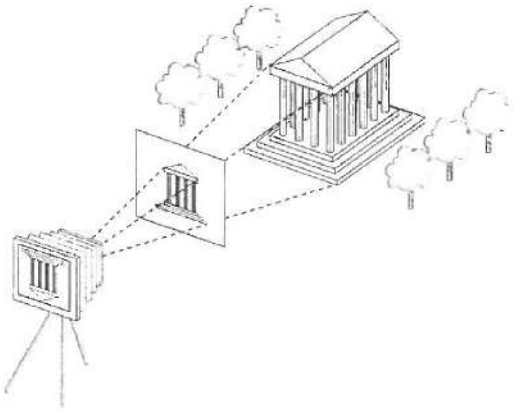


FIGURE 1.18 Imaging with the synthetic camera.

ANSWER:

(Angel pp. 20)

Our models of optical imaging systems lead directly to the conceptual foundation for modern three-dimensional computer graphics. We look at creating a computer-generated image as being similar to forming an image using an optical system. This paradigm has become known as the **synthetic-camera model**. Consider the imaging system shown in Figure 1.16. Again we see objects and a viewer. In this case, the viewer is a bellows camera.[6] The image is formed on the film plane at the back of the camera. So that we can emulate this process to create artificial images, we need to identify a few basic principles.

First, the specification of the objects is independent of the specification of the viewer. Hence, we should expect that, within a graphics library, there will be separate functions for specifying the objects and the viewer.

Second, we can compute the image using simple geometric calculations, just as we did with the pinhole camera. Consider a side view of a camera and a simple object, as shown in Figure 1.17. The view in part (a) is similar to that of the pinhole camera.

6. In a bellows camera, the front of the camera, where the lens is located, and the back of the camera, the film plane, are connected by flexible sides. Thus, we can move the back of the camera independently of the front of the camera, introducing additional flexibility in the image-formation process. We use this flexibility in Chapter 5.

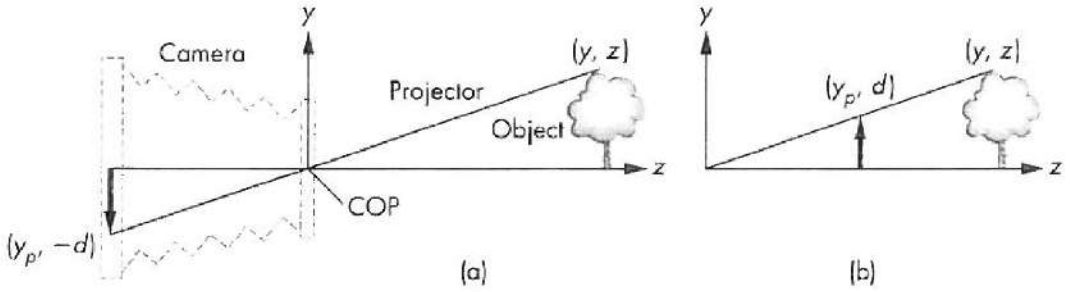


FIGURE 1.17 Equivalent views of image formation. (a) Image formed on the back of the camera. (b) Image plane moved in front of the camera.

Note that the image of the object is flipped relative to the object. Whereas with a real camera, we would simply flip the film to regain the original orientation of the object, with our synthetic camera we can avoid the flipping by a simple trick. We draw another plane in front of the lens (Figure 1.17(b)), and work in three dimensions, as shown in Figure 1.18. We find the image of a point on the object on the virtual image plane by drawing a line, called a **projector**, from the point to the center of the lens, or the **center of projection** (**COP**). Note that all projectors are rays emanating from the center of projection.

b. Define the projection plane.

ANSWER:

(Angel pp. 20)

In our synthetic camera, the virtual image plane that we have moved in front of the lens is called the **projection plane**.

c. Define the clipping rectangle or clipping window.

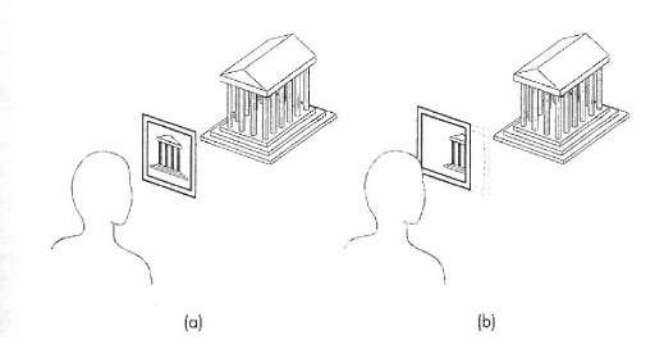


FIGURE 1.19 Clipping. (a) Window in initial position. (b) Window shifted.

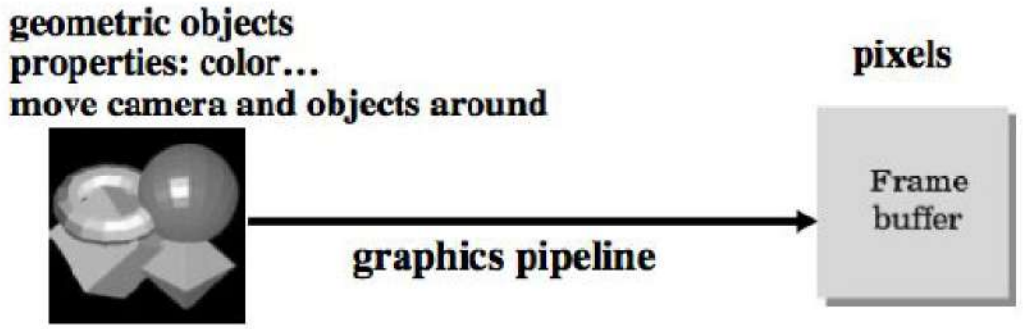
ANSWER:

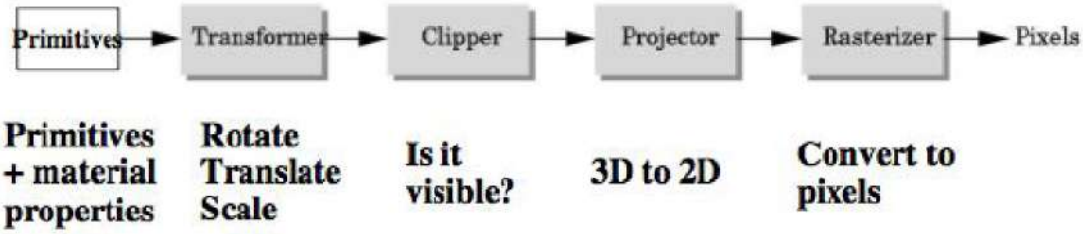
(Angel pp. 20)

In our synthetic camera, the virtual image plane that we have moved in front of the lens is called the **projection plane**. The image of the point is located where the projector passes through the projection plane. In Chapter 5, we discuss this process in detail and derive the relevant mathematical formulas.

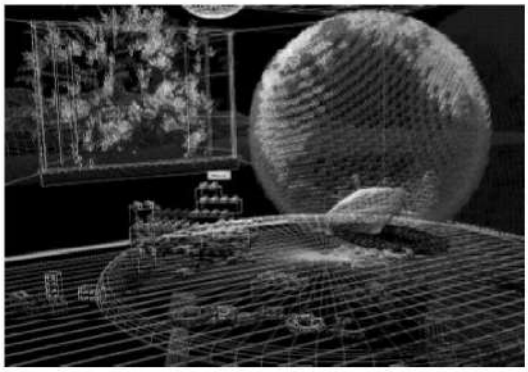
We must also consider the limited size of the image. As we saw, not all objects can be imaged onto the pinhole camera's film plane. The angle of view expresses this limitation. In the synthetic camera, we can move this limitation to the front by placing a **clipping rectangle**, or **clipping window**, in the projection plane (Figure 1.19). This rectangle acts as a window, through which a viewer, located at the center of projection, sees the world. Given the location of the center of projection, the location and orientation of the projection plane, and the size of the clipping rectangle, we can determine which objects will appear in the image.

1.6 THE PROGRAMMER'S INTERFACE





Primitives: drawing a polygon

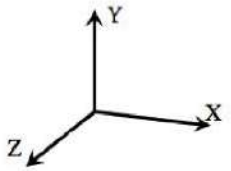


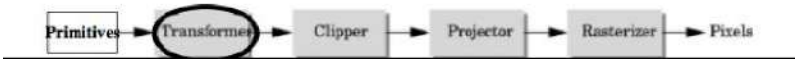
Build models in appropriate units (microns, meters, etc.).
From simple shapes: triangles, polygons,…

Transforms

- Rotate
- Translate
- Scale

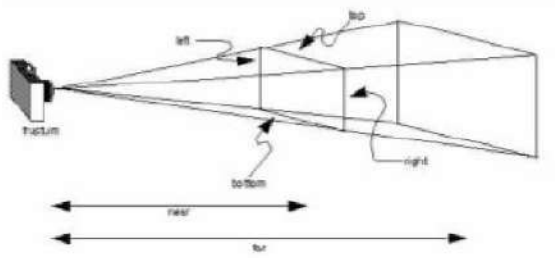
- glRotate(x,y,z);
- glTranslate(x,y,z);
- draw geometry





Clipping

Not everything should be visible on the screen

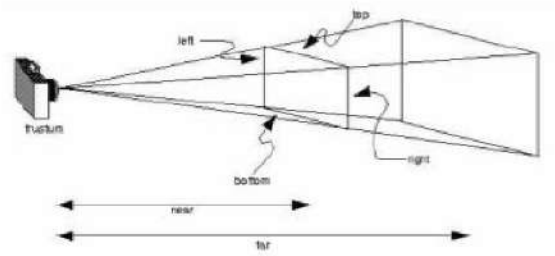


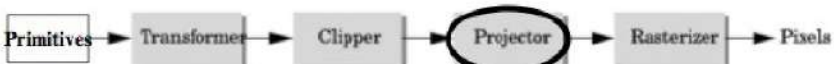
any vertices that lie outside of the viewing volume are clipped

Position it relative to the camera

Perspective projection

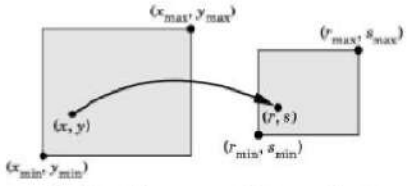
glFrustum (left, right, bottom, top, near, far);

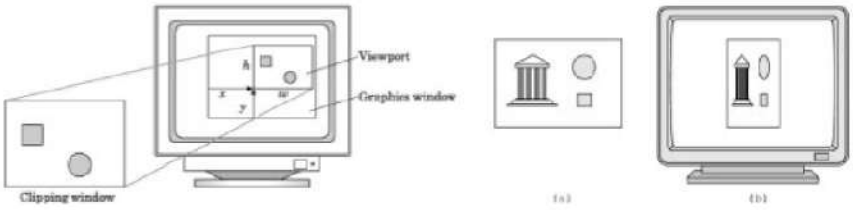


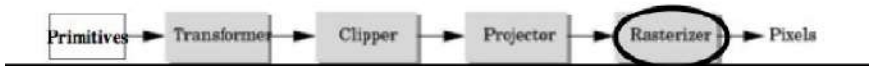


Rasterizer

Go from pixel value in world coordinates to pixel value in screen coordinates







a. Define the rasterization.

ANSWER:

(Angel pp. 24)

An alternate raster-based, but still limiting, two-dimensional model relies on writing pixels directly into a frame buffer. Such a system could be based on a single function of the form

```
write_pixel(x, y, color)
```

where `x,y` is the location of the pixel in the frame buffer and `color` gives the color to be written there. Such models are well suited to writing the algorithms for rasterization and processing of digital images.

b. Define the three-dimensional graphics systems.

ANSWER:

(Angel pp. 24)

We are much more interested, however, in the three-dimensional world. The pen-plotter model does not extend well to three-dimensional graphics systems. For example, if we wish to use the pen-plotter model to produce the image of a three-dimensional object on our two-dimensional pad, either by hand or by computer, then we have to figure out where on the page to place two-dimensional points corresponding to points on our three-dimensional object. These two-dimensional points are, as we saw in Section 1.5, the projections of points in three-dimensional space. The mathematical process of determining projections is an application of trigonometry. We develop the mathematics of projection in Chapter 5; understanding projection is crucial to understanding three-dimensional graphics. We prefer, however, to use an API that allows users to work directly in the domain of their problems and to use computers to carry out the details of the projection process automatically, without the users having to make any trigonometric calculations within the application program. That approach should be a boon to users who have difficulty learning to draw various projections on a drafting board or sketching objects in perspective. More important, users can rely on hardware and software implementations of projections within the implementation of the API that are far more efficient than any possible implementation of projections within their programs would be.

1.6.2 Three-Dimensional APIs

a. Define the Three-Dimensional APIs.

ANSWER:

(Angel pp. 24-25)

The synthetic-camera model is the basis for all the popular APIs, including OpenGL, Direct3D, and Open Scene Graph. If we are to follow the synthetic-camera model, we need functions in the API to specify the following:

- Objects
- A viewer
- Light sources
- Material properties

b. Define how are the OBJECTS defined.

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0); /* vertex A */
    glVertex3f(0.0, 1.0, 0.0); /* vertex B */
    glVertex3f(0.0, 0.0, 1.0); /* vertex C */
glEnd( );
```

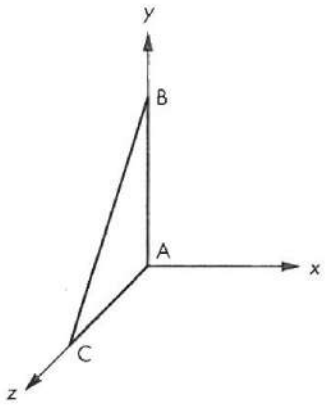
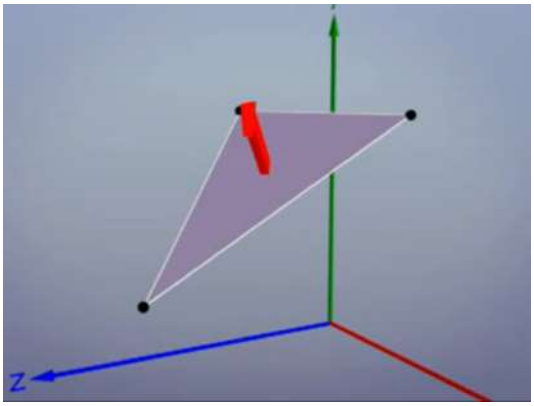


FIGURE 1.24 A triangle.

The Polygon
http://www.youtube.com/watch?v=-bZ7gstIWyI&feature=youtube_gdata



ANSWER:

(Angel pp. 25)

Objects are usually defined by sets of vertices. For simple geometric objects—such as line segments, rectangles, and polygons—there is a simple relationship between a list of **vertices**, or positions in space, and the object. For more complex objects, there may be multiple ways of defining the object from a set of vertices. A circle, for example, can be defined by three points on its circumference, or by its center and one point on the circumference.

Most APIs provide similar sets of primitive objects for the user. These primitives are usually those that can be displayed rapidly on the hardware. The usual sets include points, line segments, polygons, and sometimes text. OpenGL programs define primitives through lists of vertices. The following OpenGL code fragment specifies the triangular polygon shown in Figure 1.24 through five function calls:

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0); /* vertex A */
    glVertex3f(0.0, 1.0, 0.0); /* vertex B */
    glVertex3f(0.0, 0.0, 1.0); /* vertex C */
glEnd( );
```

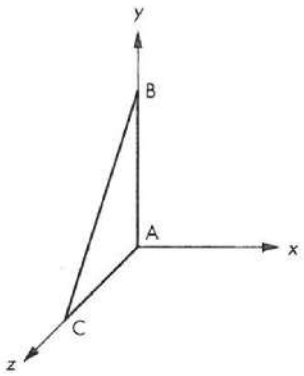


FIGURE 1.24 A triangle.

c. Define how is the VIEWER or Camera defined.

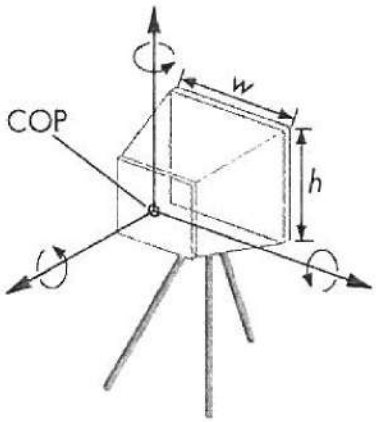


FIGURE 1.25 Camera specification.

- Viewing Transformation
  - gluLookAt

ANSWER:

(Angel pp. 25)

We can specify a viewer or camera in a variety of ways. Available APIs differ in how much flexibility they provide in camera selection and in how many different methods they allow. If we look at the camera shown in Figure 1.25, we can identify four types of necessary specifications:

1. **Position** The camera location usually is given by the position of the center of the lens, which is the center of projection (COP).
2. **Orientation** Once we have positioned the camera, we can place a camera coordinate system with its origin at the center of projection. We can then rotate the camera independently around the three axes of this system.
3. **Focal length** The focal length of the lens determines the size of the image on the film plane or, equivalently, the portion of the world the camera sees.
4. **Film plane** The back of the camera has a height and a width. On the bellows camera, and in some APIs, the orientation of the back of the camera can be adjusted independently of the orientation of the lens.

These specifications can be satisfied in various ways. One way to develop the specifications for the camera location and orientation uses a series of coordinate-system transformations. These transformations convert object positions represented in a coordinate system that specifies object vertices to object positions in a coordinate system centered at the COP. This approach is useful, both for implementing and for getting the full set of views that a flexible camera can provide. We use this approach extensively, starting in Chapter 5.

Having many parameters to adjust, however, can also make it difficult to get a desired image. Part of the problem lies with the synthetic-camera model. Classical viewing techniques, such as the ones used in architecture, stress the *relationship* between the object and the viewer, rather than the *independence* that the synthetic-camera model emphasizes. Thus, the classical two-point perspective of a cube shown in Figure 1.26 is a *two-point* perspective because of a particular relationship between the viewer and the planes of the cube (see Exercise 1.7). Although the OpenGL API allows us to set transformations with complete freedom, it also provides helpful extra functions. For example, consider the following function calls:

d. VIEWER :Define how is the gluLookAt API defined.

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
               GLdouble centerx, GLdouble centery, GLdouble centerz,
               GLdouble upx, GLdouble upy, GLdouble upz);
```

ANSWER:

(Angel pp. 26)

The first function call points the camera from a center of projection toward a desired point (the *at* point), with a specified *up* direction for the camera. The second selects a lens for a perspective view (the *field of view*) and how much of the world that the camera should image (the *aspect ratio* and the *near* and *far* distances). However, [illegible] the synthetic-camera model provide functions for directly

e VIEWER : Given the gluLookAt below in Figure 3-11

The **gluLookAt**() routine is particularly useful when you want to pan across a landscape, for instance. With a viewing volume that's symmetric in both *x* and *y*, the (*eyex*, *eyey*, *eyez*) point specified is always in the center of the image on the screen, so you can use a series of commands to move this point slightly, thereby panning across the scene.

void **gluLookAt**(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,
GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*,
GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);

Defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified by *eyex*, *eyey*, and *eyez*. The *centerx*, *centery*, and *centerz* arguments specify any point along the desired line of sight, but typically they specify some point in the center of the scene being looked at. The *upx*, *upy*, and *upz* arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).

In the default position, the camera is at the origin, is looking down the negative *z*-axis, and has the positive *y*-axis as straight up. This is the same as calling

```
gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);
```

The *z*-value of the reference point is –100.0, but could be any negative *z*, because the line of sight will remain the same. In this case, you don't actually want to call **gluLookAt**(), because this is the default (see Figure 3-11) and you are already there. (The lines extending from the camera represent the viewing volume, which indicates its field of view.)

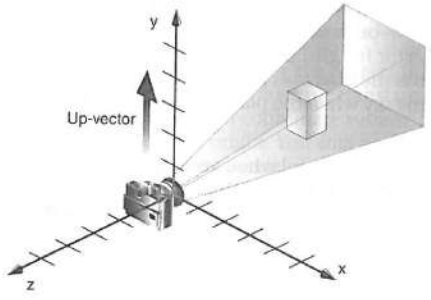


Figure 3-11 Default Camera Position

how is the gluLookAt API defined for the Figure 3-12?

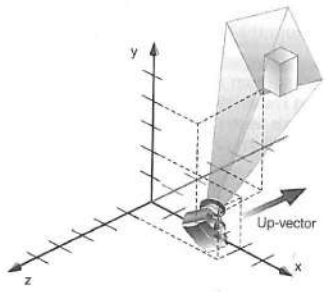


Figure 3-12 Using gluLookAt()

ANSWER:

(Angel pp. 26)

Figure 3-12 shows the effect of a typical **gluLookAt()** routine. The camera position (*eyex, eyey, eyez*) is at (4, 2, 1). In this case, the camera is looking right at the model, so the reference point is at (2, 4, –3). An orientation vector of (2, 2, –1) is chosen to rotate the viewpoint to this 45-degree angle.

Therefore, to achieve this effect, call

```
gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

f. VIEWER : Define how is the gluPerspective API defined.

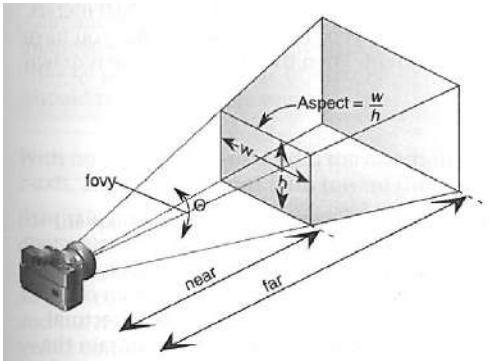


Figure 3-14 Perspective Viewing Volume Specified by gluPerspective()

void **gluPerspective**(GLdouble ***fovy***, GLdouble ***aspect***, GLdouble ***zNear***, GLdouble ***zFar***);

```
void gluPerspective(GLdouble fovy, GLdouble aspect,
                    GLdouble near, GLdouble far);
```

ANSWER:

(Angel pp. 26)
Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. The *fovy* argument is the angle of the field of view in the *x*-*z* plane; its value must be in the range [0.0,180.0]. The *aspect* ratio is the width of the frustum divided by its height. The *zNear* and *zFar* values are the distances between the viewpoint and the clipping plane

. The second selects
a lens for a perspective view (the *field of view*) and how much of the world that
the camera should image (the *aspect ratio* and the *near* and *far* distances

g. VIEWER : Define how is the glOrtho API defined.

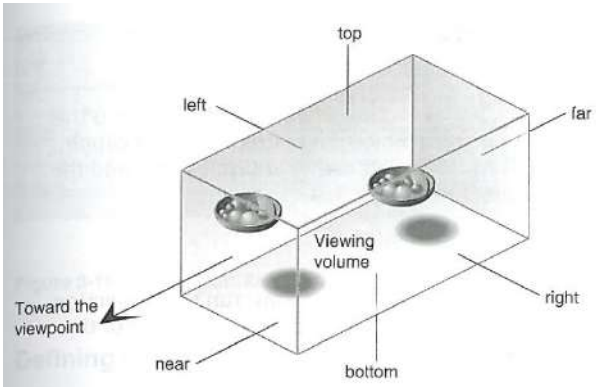


Figure 3-15 Orthographic Viewing Volume

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
             GLdouble top, GLdouble near, GLdouble far);
```

ANSWER:

Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (*left, bottom, –near*) and (*right, top, –near*) are points on the near clipping plane that are mapped to the lower left and upper right corners of the viewport window, respectively. (*left, bottom, –far*) and (*right, top, –far*) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both *near* and *far* may be positive, negative, or even set to zero. However, *near* and *far* should not be the same value.

With no other transformations, the direction of projection is parallel to the *z*-axis, and the viewpoint faces toward the negative *z*-axis.

h. Define how is the LIGHT sources are specified.

- Setting light intensities:

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, Id);
glLightfv(GL_LIGHT0, GL_SPECULAR, Is);
```

ANSWER:

(Angel pp. 26)

Light sources are defined by their location, strength, color, and directionality. APIs provide a set of functions to specify these parameters for each source

i. Define how is the MATERIAL properties of an object are specified.

- Setting material colors:

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, Ka);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, Kd);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, Ks);
```

ANSWER:

(Angel pp. 26)

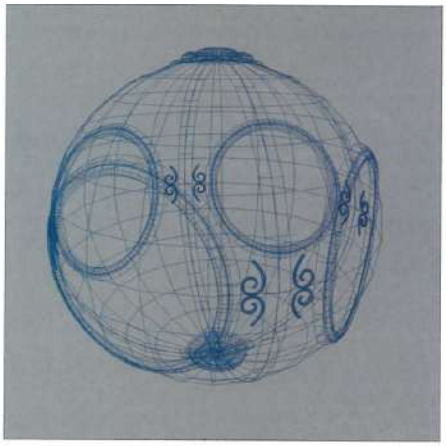
. Material properties are characteristics, or attributes, of the objects, and such properties are specified through a series of function calls at the time that each object is defined. Both light sources and material properties depend on the models of light–material interactions supported by the API. We discuss such models in Chapter 6.

1.6.3 A Sequence of Images

a. Define wire-frame image.

Color Plate 2 Wire-frame representation of sun object surfaces.

(Courtesy of Full Dome Project, University of New Mexico.)



ANSWER:

(Angel pp. 27)

object shows only the outlines of the parts. This type of image is known as a **wire-frame** image because we can see only the edges of surfaces: Such an image would be produced if the objects were constructed with stiff wires that formed a frame with no solid material between the edges. Before raster-graphics systems became available, wireframe images were the only type of computer-generated images that we could produce

b. Define rendered with flat polygons.

Color Plate 3 Flat-shaded polygonal rendering of sun object.

(Courtesy of Full Dome Project, University of New Mexico.)

ANSWER:

(Angel pp. 27)

Most raster systems can fill the interior of polygons with a solid color in approximately the same time that they can render a wireframe image. Although the objects are three dimensional, each surface is displayed in a single color, and the image fails to show the three-dimensional shapes of the objects. Early raster systems could produce images of this form.

c. Define smooth shading of a polygon.

Color Plate 4 Smooth-shaded polygonal rendering of sun object.

(Courtesy of Full Dome Project, University of New Mexico.)

ANSWER:

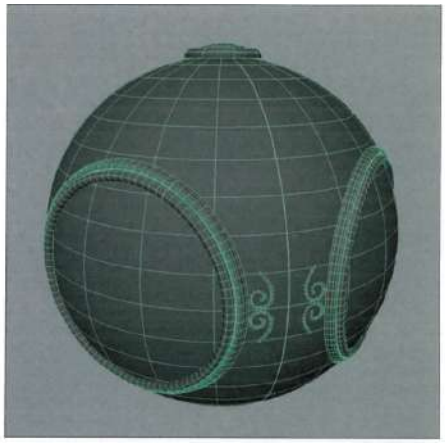
(Angel pp. 27)

Color Plate 4 illustrates smooth shading of the polygons that approximate the object; it shows that the object is three dimensional and gives the appearance of a smooth surface. We develop shading models that are supported by OpenGL in Chapter 6. These shading models are also supported in the hardware of most recent workstations; generating the shaded image on one of these systems takes approximately the same amount of time as does generating a wireframe image.

d. Define NURBS surfaces.

Color Plate 5 Wire-frame of NURBS representation of sun object showing the high number of polygons used in rendering the NURBS surfaces.

(Courtesy of Full Dome Project, University of New Mexico.)



ANSWER:

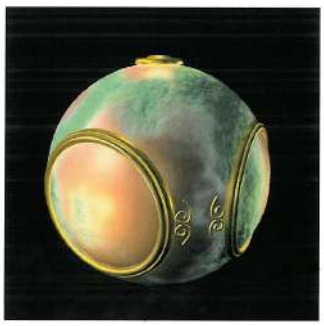
(Angel pp. 27)

Color Plate 5 shows a more sophisticated wireframe model constructed using NURBS surfaces, which we introduce in Chapter 12. Such surfaces give the application programmer great flexibility in the design process but are ultimately rendered using line segments and polygons.

e. Define surface texture.

Color Plate 6 Rendering of sun object showing bump map.

(Courtesy of Full Dome Project, University of New Mexico.)



Color Plate 7 Rendering of sun object with an environment map.

(Courtesy of Full Dome Project, University of New Mexico.)

ANSWER:

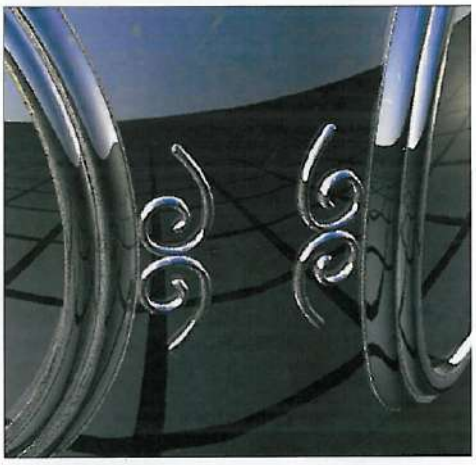
(Angel pp. 27)

In Color Plates 6 and 7, we add surface texture to our object; texture is one of the effects that we discuss in Chapter 8. All recent graphics processors support texture mapping in hardware, so rendering of a texture-mapped image requires little additional time. In Color Plate 6, we use a technique called *bump mapping* that gives the appearance of a rough surface even though we render the same flat polygons as in the other examples. Color Plate 7 shows an *environment map* applied to the surface of the object, which gives the surface the appearance of a mirror. These techniques will be discussed in detail in Chapters 8 and 9

f. Define antialiasing.

Color Plate 8 Rendering of a small part of the sun object with an environment map. (Courtesy of Full Dome Project, University of New Mexico.)

(a) Without antialiasing



(b) With antialiasing

ANSWER:

(Angel pp. 28)

Color Plate 8 shows a small area of the rendering of the object using an environment map. The image on the left shows the jagged artifacts known as aliasing errors that are due to the discrete nature of the frame buffer. The image on the right has been rendered using a smoothing or antialiasing method that we will study in Chapters 7 and 8.

1.6.4 The Modeling-Rendering Paradigm

a. Define the separation of modeling from rendering of the scene.

FIGURE 1.27 The modeling–rendering pipeline.

ANSWER:

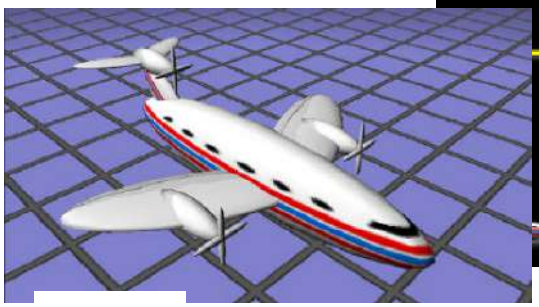
(Angel pp. 28)

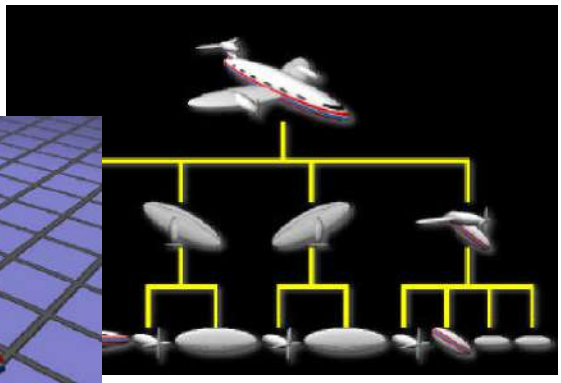
In many situations—especially in CAD applications and in the development of complex images, such as for movies—we can separate the modeling of the scene from the production of the image, or the **rendering** of the scene. Hence, we can look at image formation as the two-step process shown in Figure 1.27. Although the tasks are the same as those we have been discussing, this block diagram suggests that we might implement the modeler and the renderer with different software and hardware. For example, consider the production of a single frame in an animation. We first want to design and position our objects. This step is highly interactive, and we do not need to work with detailed images of the objects. Consequently, we prefer to carry out this step on an interactive workstation with good graphics hardware. Once we have designed the scene, we want to render it, adding light sources, material properties, and a variety of other detailed effects, to form a production-quality image. This step requires a tremendous amount of computation, so we prefer to use a high-performance cluster or a render farm. Not only is the optimal hardware different in the modeling and rendering steps, but the software that we use also may be different.

The interface between the modeler and renderer can be as simple as a file produced by the modeler that describes the objects and that contains additional information important only to the renderer, such as light sources, viewer location, and material properties. Pixar's RenderMan interface follows this approach and uses a file

format that allows modelers to pass models to the renderer in text format. One of the other advantages of this approach is that it allows us to develop modelers that, although they use the same renderer, are tailored to particular applications. Likewise, different renderers can take as input the same interface file. It is even possible, at least in principle, to dispense with the modeler completely and to use a standard text editor to generate an interface file. For any but the simplest scenes, however, users cannot edit lists of information for a renderer. Rather, they use interactive modeling software. Because we must have at least a simple image of our objects to interact with a modeler, most modelers use the synthetic-camera model to produce these images in real time.

b. Define the scene graph.





real time.

This paradigm has become popular as a method for generating computer games and images over the Internet. Models, including the geometric objects, lights, cameras, and material properties, are placed in a data structure called a **scene graph** that is passed to a renderer or a game engine. We will examine scene graphs in Chapter 10. It is also the standard method used in the animation industry where interactive programs such as Maya and Lightwave are used interactively to build characters using wireframes or unlit polygons. Final rendering can take hours per frame.

1.7.2 Pipeline Architectures

a. Define pipelining, latency and throughput.

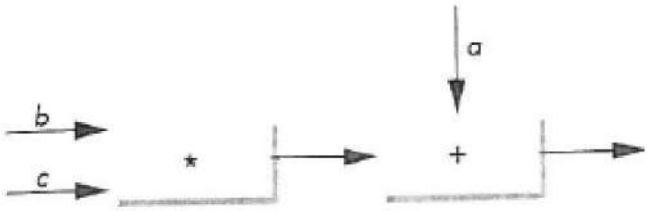


FIGURE 1.30 Arithmetic pipeline.

ANSWER:

(Angel pp. 30)

Pipelining is similar to an assembly line in a car plant. As the chassis passes down the line, a series of operations is performed on it, each using specialized tools and workers, until at the end, the assembly process is complete. At any one time, multiple cars are under construction and there is a significant delay or **latency** between when a chassis starts down the assembly line and the finished vehicle is complete. However, the number of cars produced in a given time, the **throughput**, is much higher than if a single team built each car.

The concept of pipelining is illustrated in Figure 1.30 for a simple arithmetic calculation. In our pipeline, there is an adder and a multiplier. If we use this configuration to compute $a + (b * c)$, then the calculation takes one multiplication and one addition—the same amount of work required if we use a single processor to carry out both operations. However, suppose that we have to carry out the same computation with many values of a, b, and c. Now, the multiplier can pass on the results of its calculation to the adder and can start its next multiplication while the adder carries out the second step of the calculation on the first set of data. Hence, whereas it takes the same amount of time to calculate the results for any one set of data, when we are working on two sets of data at one time, our total time for calculation is shortened markedly. Here the rate at which data flows through the system, the throughput of the system, has been doubled. Note that as we add more boxes to a pipeline, the latency of the system increases and we must balance latency against increased throughput in evaluating the performance of a pipeline.

We can construct pipelines for more complex arithmetic calculations that will afford even greater increases in throughput. Of course, there is no point in building a pipeline unless we will do the same operation on many data sets. But that is just what we do in computer graphics, where large sets of vertices must be processed in the same manner.

1.7.3 The Graphics Pipeline

a. Define geometry of the scene.

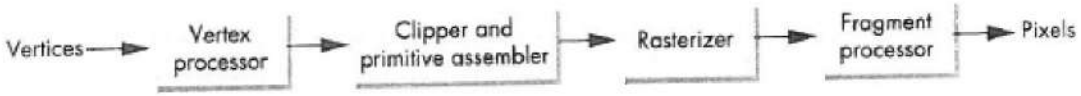


FIGURE 1.31 Geometric pipeline.

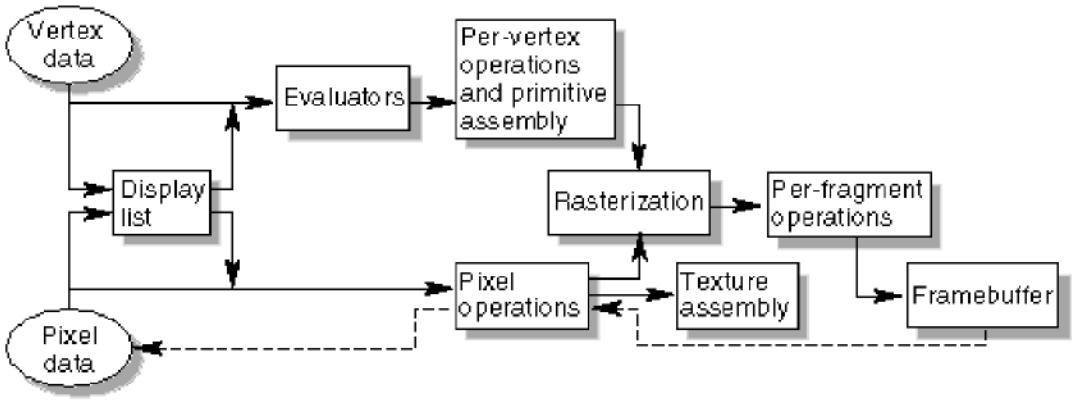


Figure 1-2 : Order of Operations

ANSWER:

(Angel pp. 31)

We start with a set of objects. Each object comprises a set of graphical primitives. Each primitive comprises a set of vertices. We can think of the collection of primitive types and vertices as defining the **geometry** of the scene. In a complex scene, there may be thousands—even millions—of vertices that define the objects. We must process all these vertices in a similar manner to form an image in the frame buffer. If we think in terms of processing the geometry of our objects to obtain an image, we can employ the block diagram in Figure 1.31

b. Define the four steps in the imaging process.

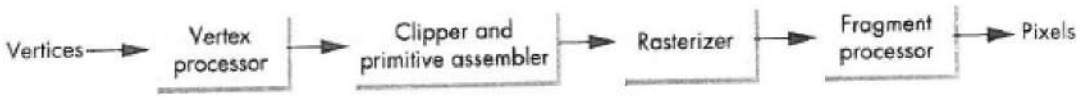


FIGURE 1.31 Geometric pipeline.

ANSWER:

(Angel pp. 31)

1. Vertex processing
2. Clipping and primitive assembly
3. Rasterization
4. Fragment processing

1.7.4 The Vertex Processing

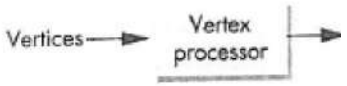
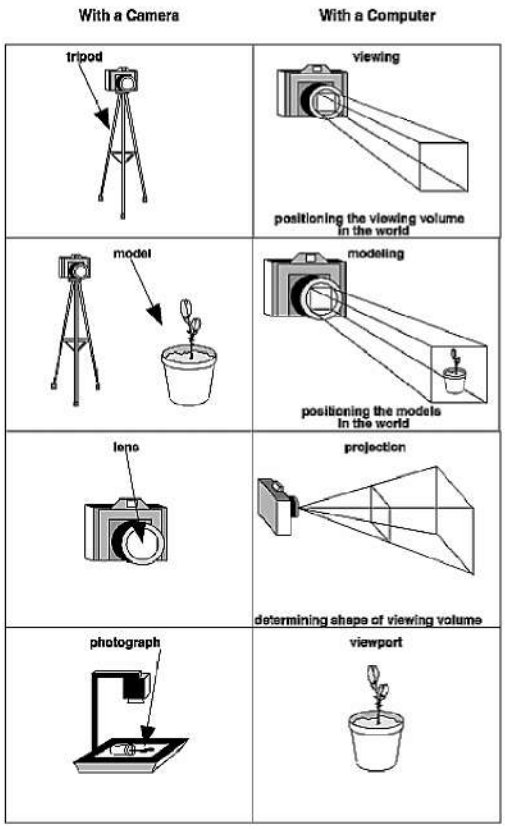


FIGURE 1.31 Geometric pipeline.

a. Define how vertices are processed and what is processed.



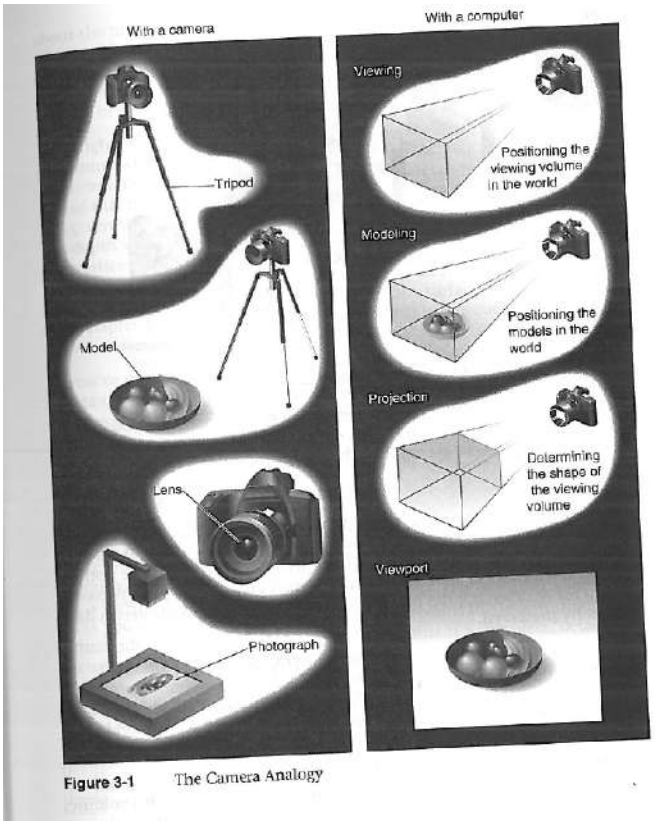


Figure 3-1 The Camera Analogy

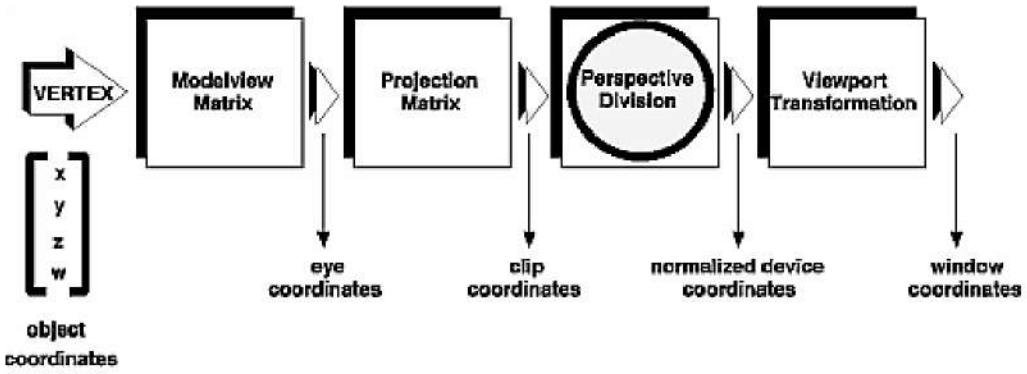


Figure 3-2 : Stages of Vertex Transformation

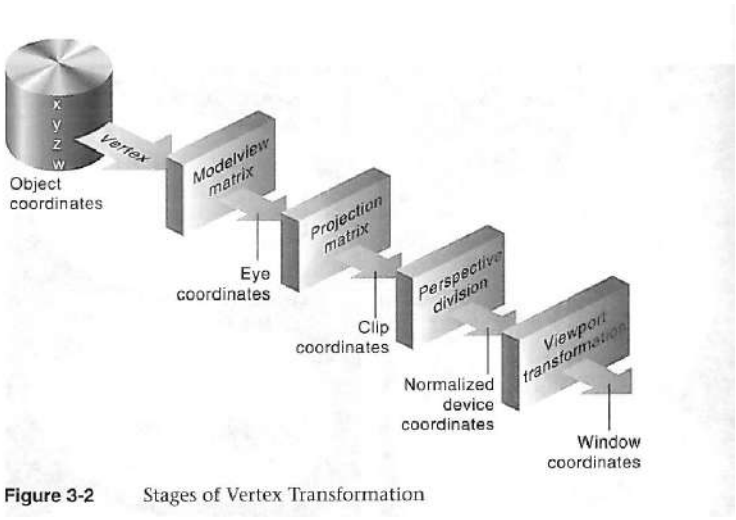


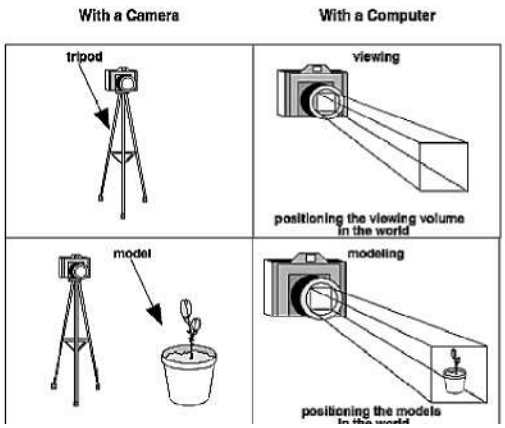
Figure 3-2 Stages of Vertex Transformation

ANSWER:

(Angel pp. 32)

In the first block of our pipeline, each vertex is processed independently. The two major functions of this block are to carry out coordinate transformations and to compute a color for each vertex.

b. Define modeling (OBJECT) & viewing (CAMERA) transformations.



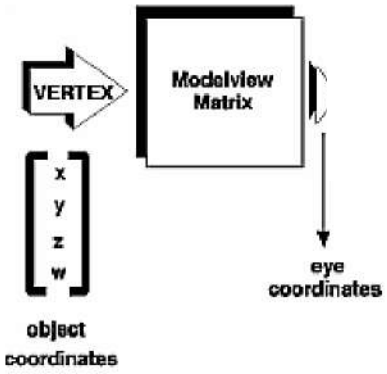


Figure 3-2 : Model View Transformations

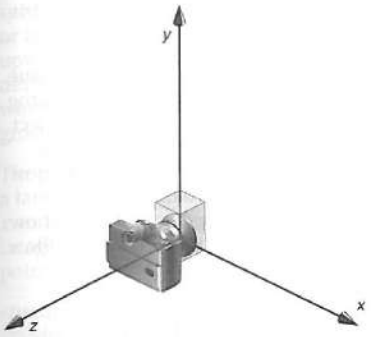


Figure 3-9 Object and Viewpoint at the Origin

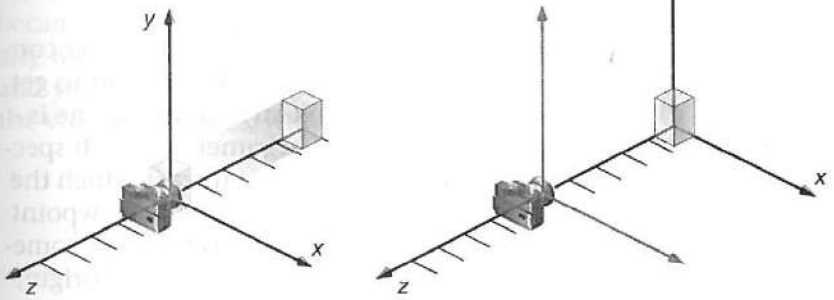


Figure 3-10 Separating the Viewpoint and the Object

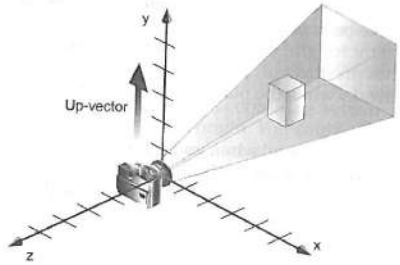


Figure 3-11 Default Camera Position

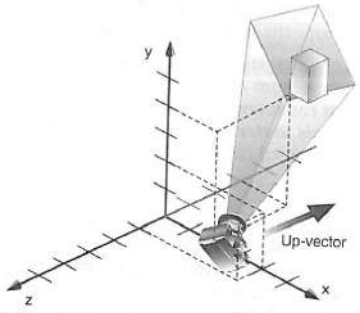


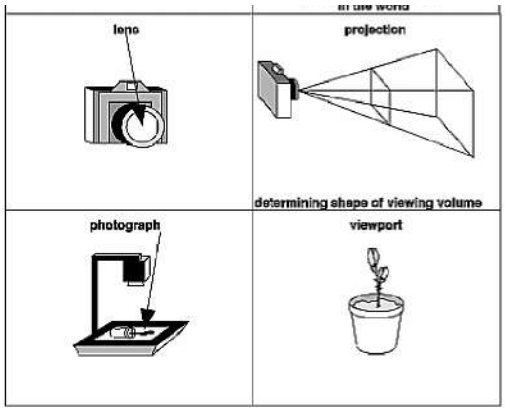
Figure 3-12 Using gluLookAt()

ANSWER:

(Angel pp. 32)

Many of the steps in the imaging process can be viewed as transformations between representations of objects in different coordinate systems. For example, in the synthetic camera paradigm, a major part of viewing is to convert to a representation of objects from the system in which they were defined to a representation in terms of the coordinate system of the camera. A further example of a transformation arises when we finally put our images onto the output device. The internal representation of objects—whether in the camera coordinate system or perhaps in a system used by the graphics software—eventually must be represented in terms of the coordinate system of the display. We can represent each change of coordinate systems by a matrix. We can represent successive changes in coordinate systems by multiplying, or **concatenating**, the individual matrices into a single matrix. In Chapter 4, we examine these operations in detail. Because multiplying one matrix by another matrix yields a third matrix, a sequence of transformations is an obvious candidate for a pipeline architecture. In addition, because the matrices that we use in computer graphics will always be small (4×4), we have the opportunity to use parallelism within the transformation blocks in the pipeline.

c. Define projection transformations.



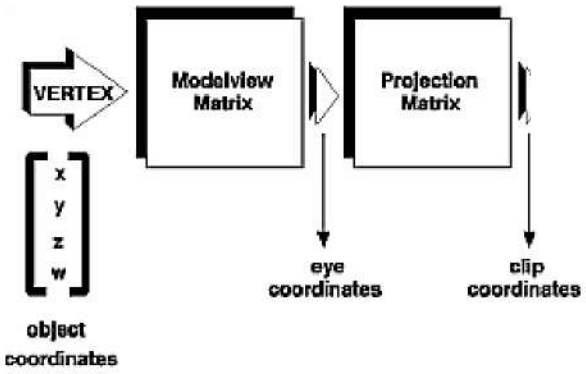


Figure 3-2 : Projection Transformations

ANSWER:

(Angel pp. 32)

Eventually, after multiple stages of transformation, the geometry is transformed by a projection transformation. In Chapter 5, we see that we can implement this step using 4 × 4 matrices, and thus projection fits in the pipeline. In general, we want to keep three-dimensional information as long as possible, as objects pass through the pipeline. Consequently, the projection transformation is somewhat more general than the projections in Section 1.5. In addition to retaining three-dimensional information, there is a variety of projections that we can implement. We will see these projections in Chapter 5.

The assignment of vertex colors can be as simple as the program specifying a color or as complex as the computation of a color from a physically realistic lighting model that incorporates the surface properties of the object and the characteristic light sources in the scene. We will discuss lighting models in Chapter 6.

d. Define viewport transformation.

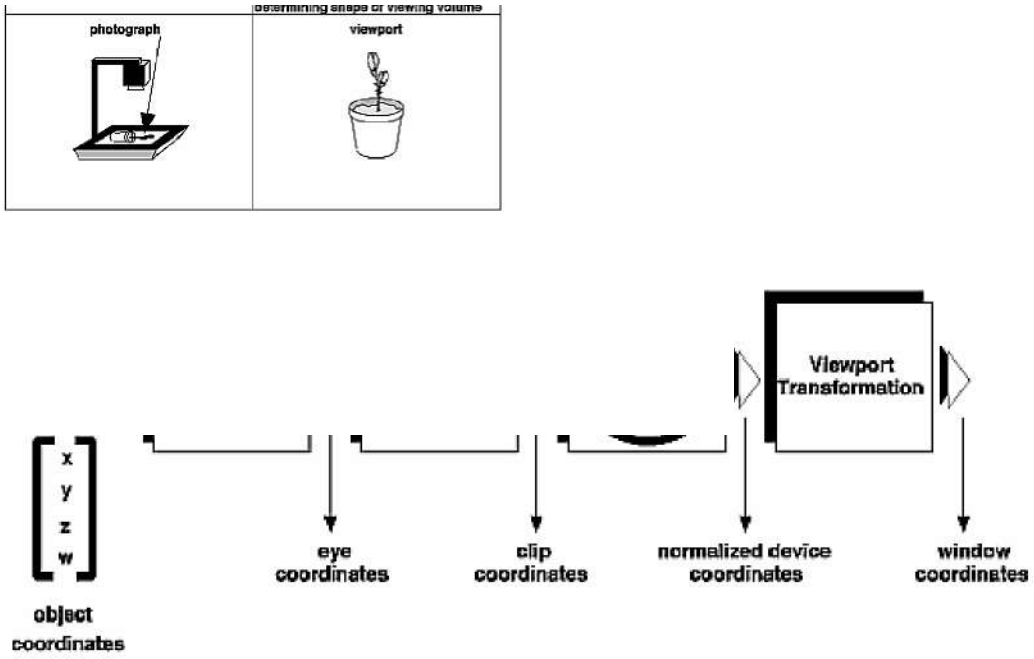


Figure 3-2 : Viewport Transformation

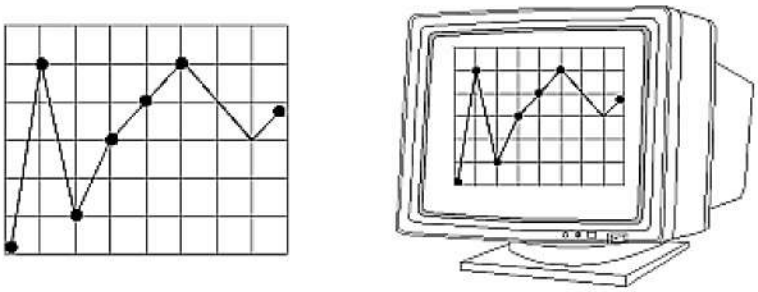


Figure 3-16 : Viewport Rectangle

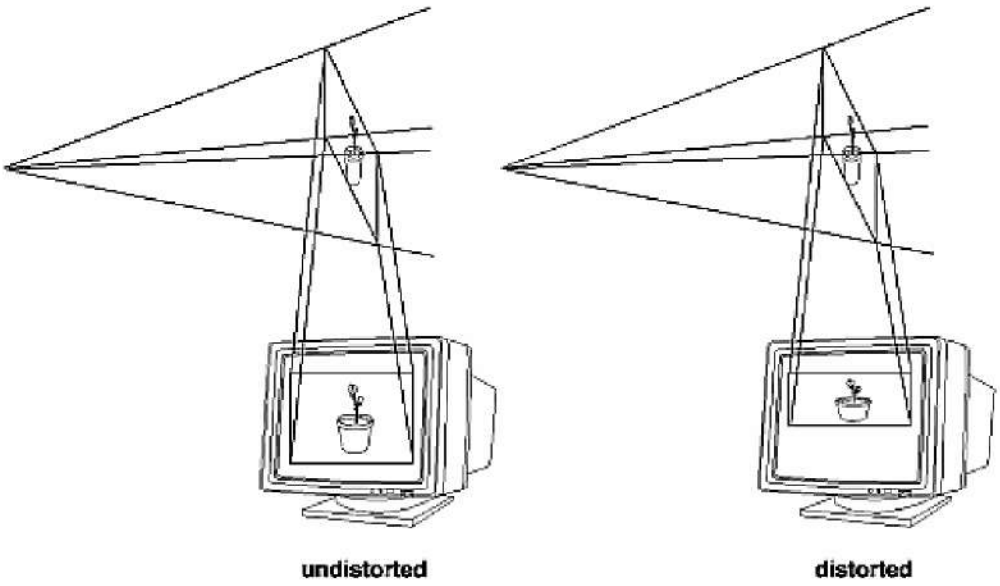


Figure 3-17 : Mapping the Viewing Volume to the Viewport

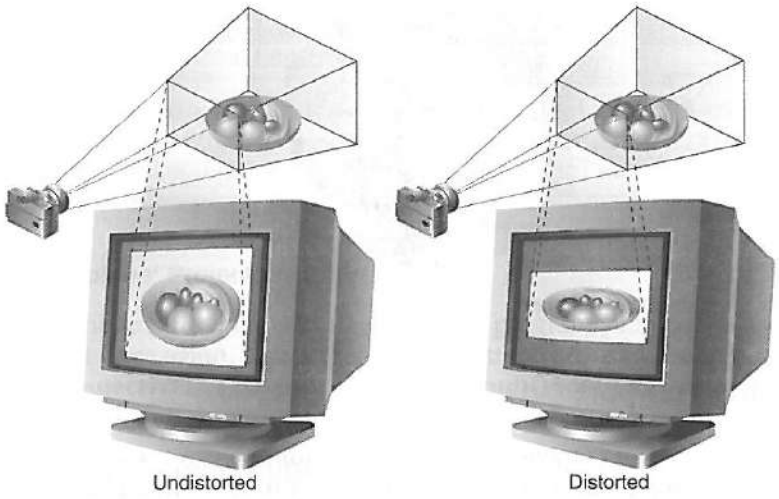
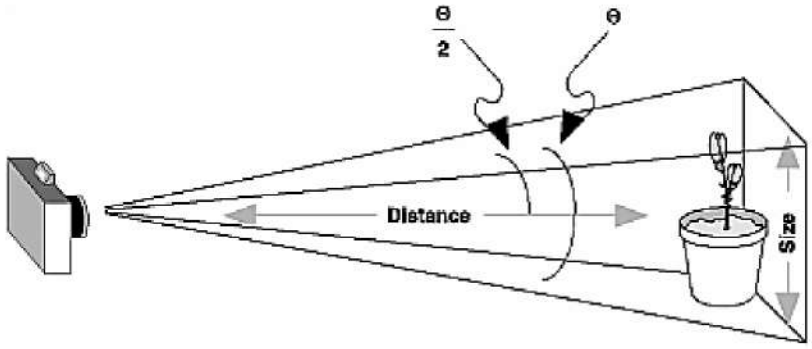


Figure 3-17 Mapping the Viewing Volume to the Viewport

*void **glViewport**(GLint x, GLint y, GLsizei width, GLsizei height);*



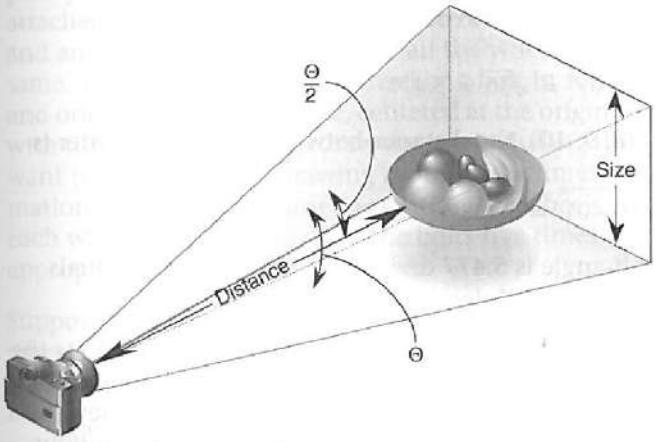


Figure 3-19 Using Trigonometry to Calculate the Field of View

Figure 3-19 : Using Trigonometry to Calculate the Field of View

ANSWER:

(Angel pp. 32)

Eventually, after multiple stages of transformation, the geometry is transformed by a projection transformation. In Chapter 5, we see that we can implement this step using 4 × 4 matrices, and thus projection fits in the pipeline. In general, we want to keep three-dimensional information as long as possible, as objects pass through the pipeline. Consequently, the projection transformation is somewhat more general than the projections in Section 1.5. In addition to retaining three-dimensional information, there is a variety of projections that we can implement. We will see these projections in Chapter 5.

1.7.5 Clipping and Primitive Assembly

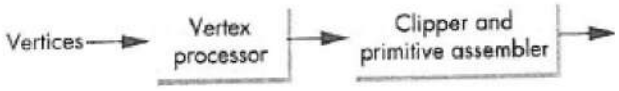


FIGURE 1.31 Geometric pipeline.

a. Define the clipping volume.

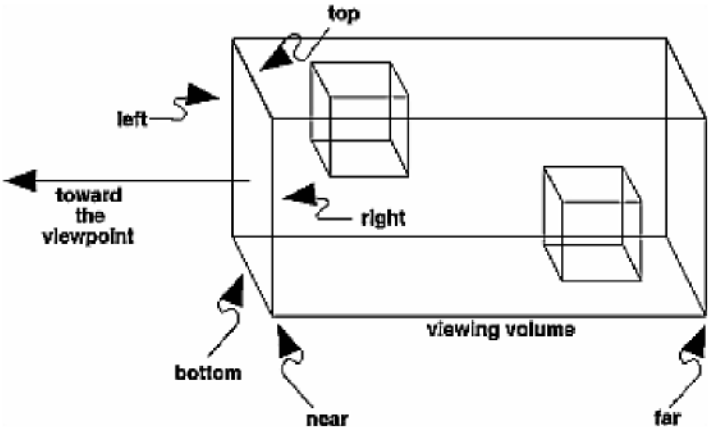


Figure 3-15 : Orthographic Viewing Volume

void **gluOrtho2D**(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

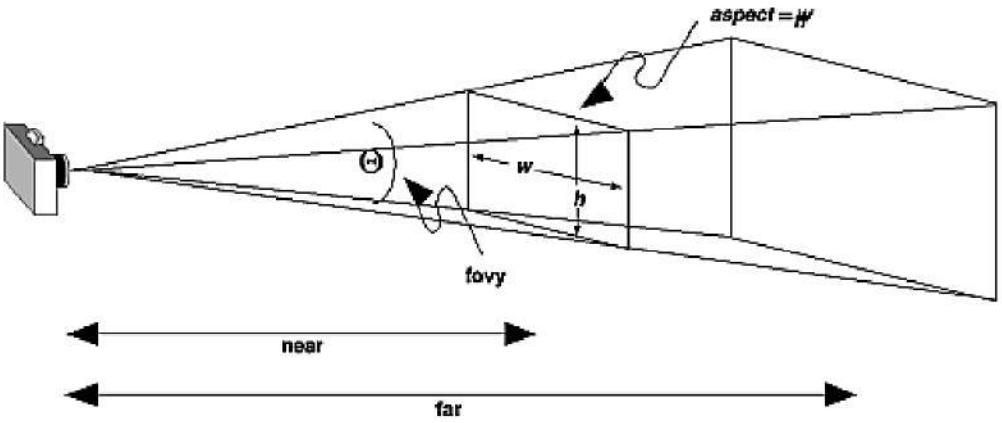


Figure 3-14 : Perspective Viewing Volume Specified by gluPerspective()

*void **gluPerspective**(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);*

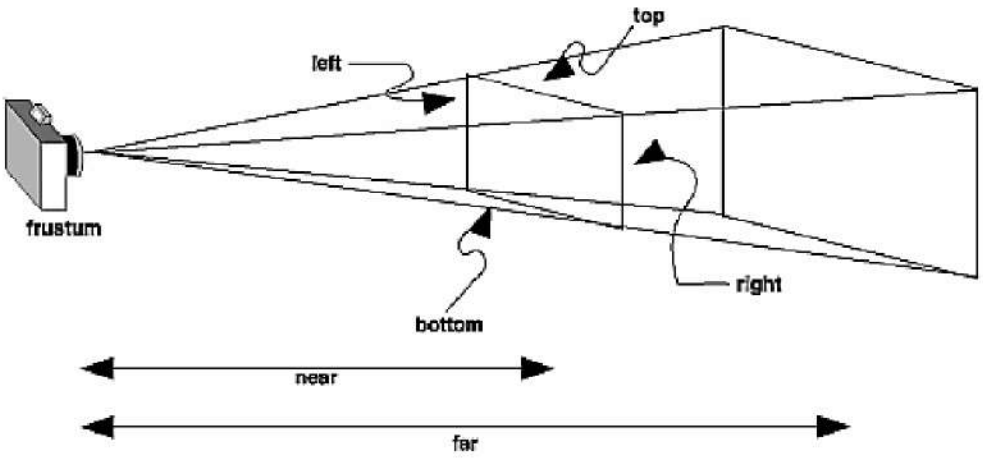


Figure 3-13 : Perspective Viewing Volume Specified by glFrustum()

*void **glFrustum**(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);*

ANSWER:

(Angel pp. 32)

The second fundamental block in the implementation of the standard graphics pipeline is for clipping and primitive assembly. We must do clipping because of the limitation that no imaging system can see the whole world at once. The human retina has a limited size corresponding to an approximately 90-degree field of view. Cameras have film of limited size, and we can adjust their fields of view by selecting different lenses.

We obtain the equivalent property in the synthetic camera by considering a **clipping volume**, such as the pyramid in front of the lens in Figure 1.18. The projections of objects in this volume appear in the image. Those that are outside do not and are said to be clipped out. Objects that straddle the edges of the clipping volume are partly visible in the image. Efficient clipping algorithms are developed in Chapter 7.

Clipping must be done on a primitive by primitive basis rather than on a vertex by vertex basis. Thus, within this stage of the pipeline, we must assemble sets of vertices into primitives, such as line segments and polygons, before clipping can take place. Consequently, the output of this stage is a set of primitives whose projections can appear in the image.

1.7.6 Rasterization

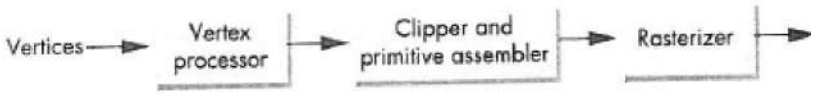


FIGURE 1.31 Geometric pipeline.

a. Define fragments.

ANSWER:

(Angel pp. 32)

The primitives that emerge from the clipper are still represented in terms of their vertices and must be further processed to generate pixels in the frame buffer. For example, if three vertices specify a triangle filled with a solid color, the rasterizer must determine which pixels in the frame buffer are inside the polygon. We discuss this rasterization (or scan-conversion) process in Chapter 8 for line segments and polygons. The output of the rasterizer is a set of **fragments** for each primitive. A fragment can be thought of as a potential pixel that carries with it information, including its color and location, that is used to update the corresponding pixel in the frame buffer. Fragments can also carry along depth information that allows later stages to determine if a particular fragment lies behind other previously rasterized fragments for a given pixel.

1.7.7 Fragment Processing

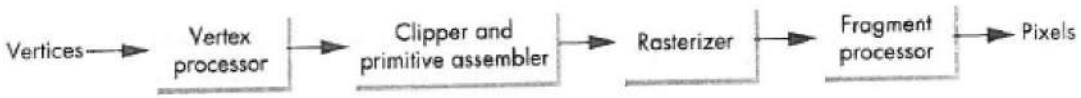


FIGURE 1.31 Geometric pipeline.

a. Describe how to determine the color of fragments.

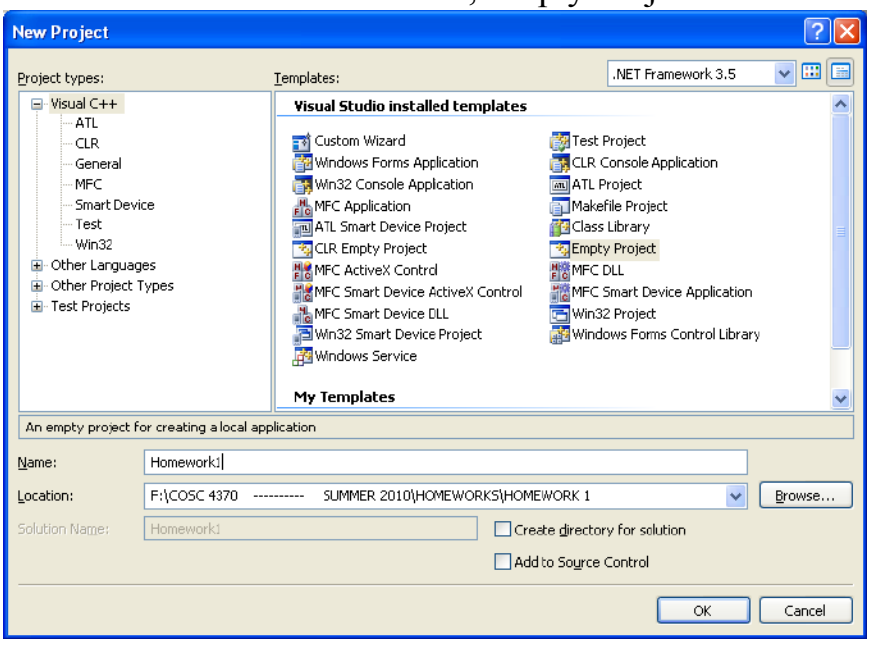
ANSWER:

(Angel pp. 33)

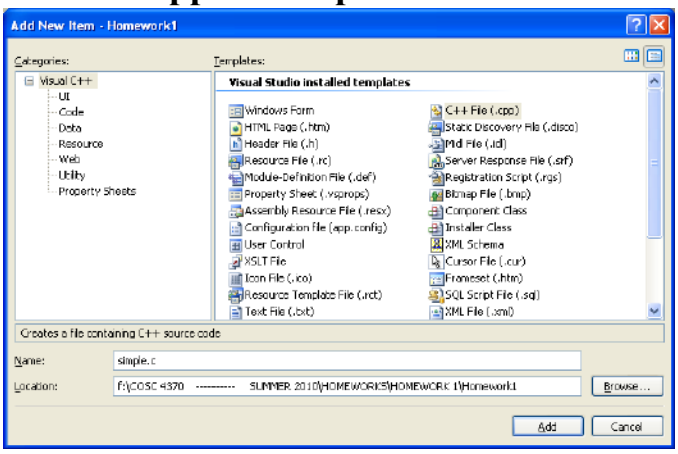
The final block in our pipeline takes in the fragments generated by the rasterizer and updates the pixels in the frame buffer. If the application generated three-dimensional data, some fragments may not be visible because the surfaces that they define are behind other surfaces. The color of a fragment may be altered by texture mapping or bump mapping as shown in Color Plates 6 and 7. The color of the pixel that corresponds to a fragment can also be read from the frame buffer and blended with the fragment's color to create translucent effects. These effects will be covered in Chapters 8 and 9.

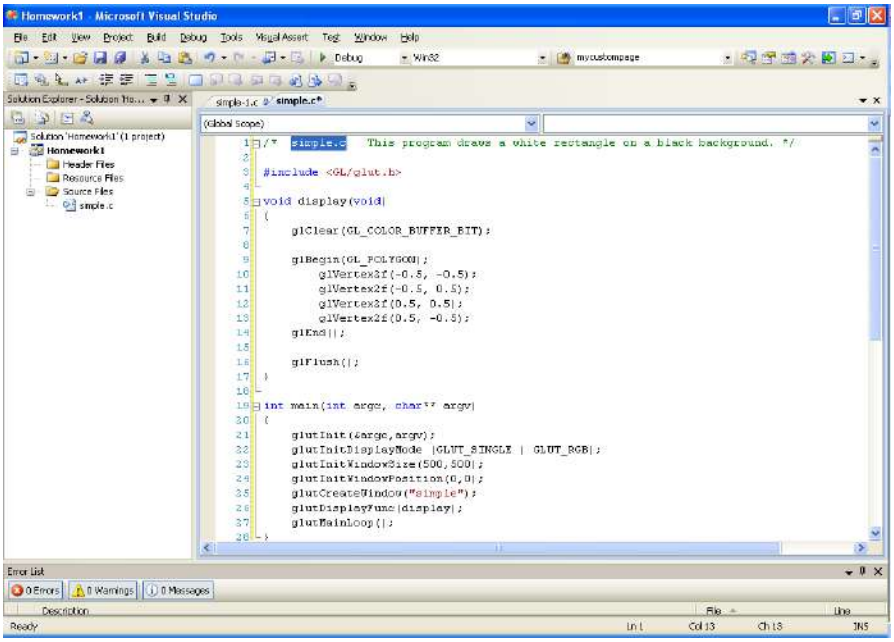
B. (100 pts) Visual Studio 2008 C++ Project

Create Visual Studio 2008 C++, Empty Project:



Create a .cpp file simple.c with the content:

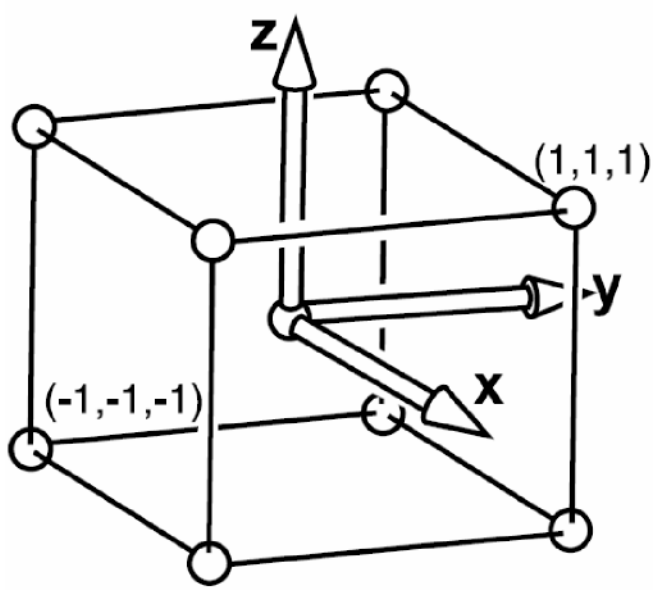




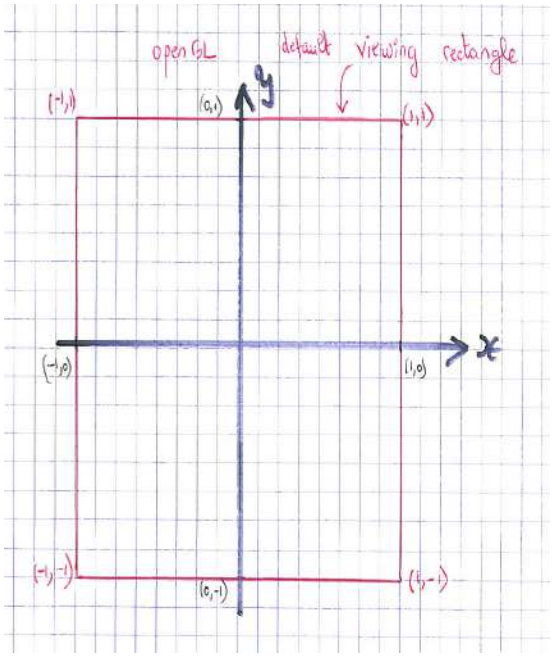
simple.c

```
/* simple.c  This program draws a white rectangle on a black background.
*/

#include <GL/glut.h>          /* glut.h includes gl.h and glu.h*/

void display(void)
{
       glClear(GL_COLOR_BUFFER_BIT);

       glBegin(GL_POLYGON);
            glVertex2f(-0.5, -0.5);
            glVertex2f(-0.5, 0.5);
            glVertex2f(0.5, 0.5);
            glVertex2f(0.5, -0.5);
       glEnd();

       glFlush();
}

int main(int argc, char** argv)
{
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(0,0);
      glutCreateWindow("simple");
      glutDisplayFunc(display);
      glutMainLoop();
}
```

The default viewing volume in openGL is presented below, with the center at (0,0,0). Thus, objects have to be inside this volume in order to be displayed:

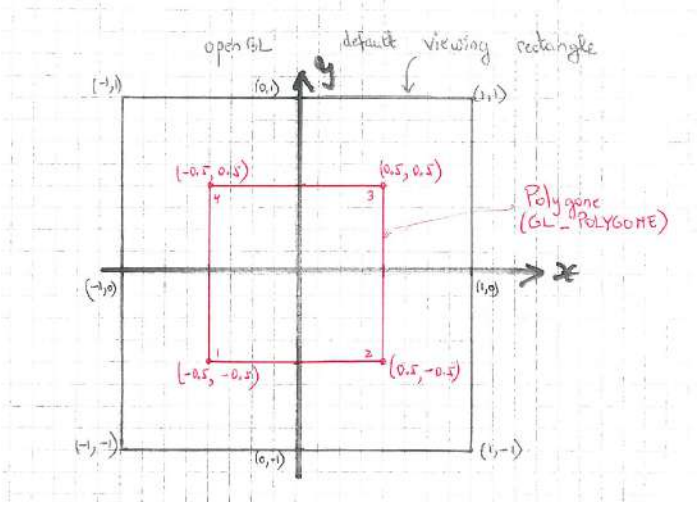


In 2D, z=0 thus the viewing volume become a viewing rectangle.
Thus, the rectangle below has x coordinates inside (-1,1) and y coordinates inside (-1,1):



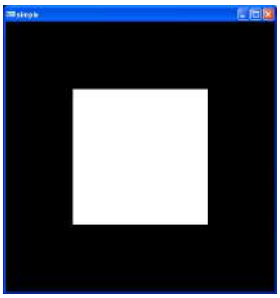
```
glBegin(GL_POLYGON);
   glVertex2f(-0.5, -0.5);
   glVertex2f(-0.5,  0.5);
   glVertex2f( 0.5,  0.5);
   glVertex2f( 0.5, -0.5);
glEnd();
```

When rendered, the polygon should look like this:



1. Build and run this Project: Insert a screenshot of your output.

ANSWER:



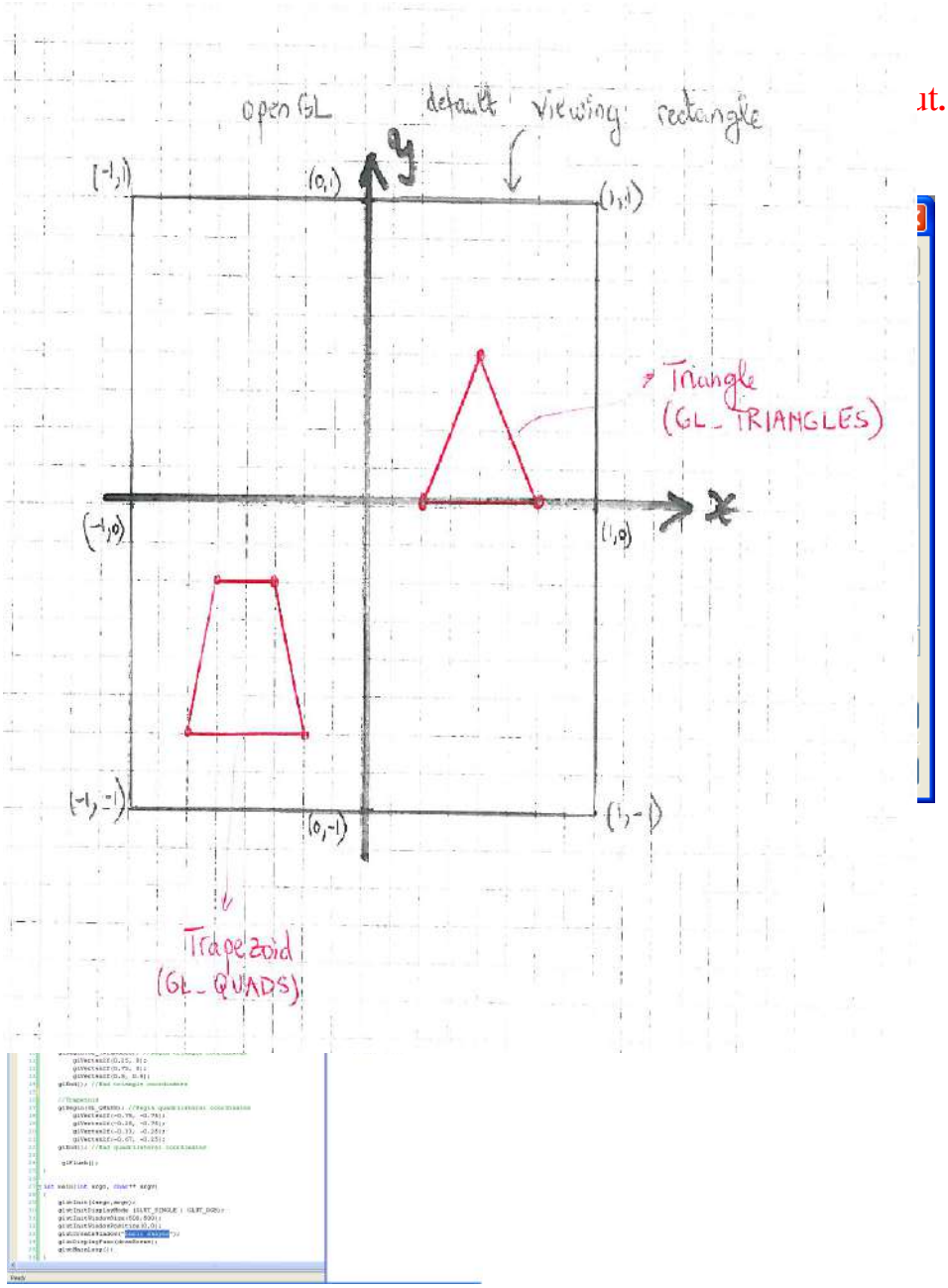
2. Create basicshapes.c from simple.c from the Homework1 Project. Modify basicshapes.c to:

1. change the window title from "simple" to "basic shapes"; Build

2. rename the display function drawScene; Build

3. inside drawScene:

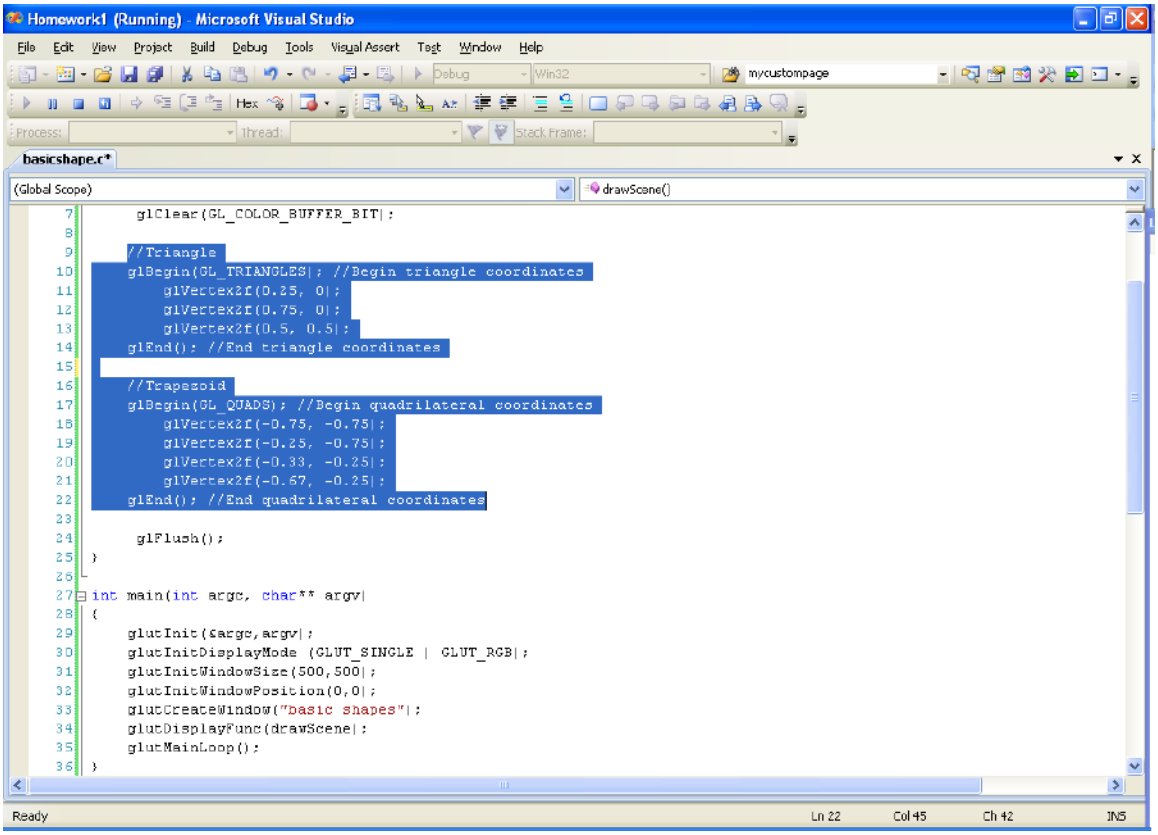
Draw the triangle and the trapezoid specified below:

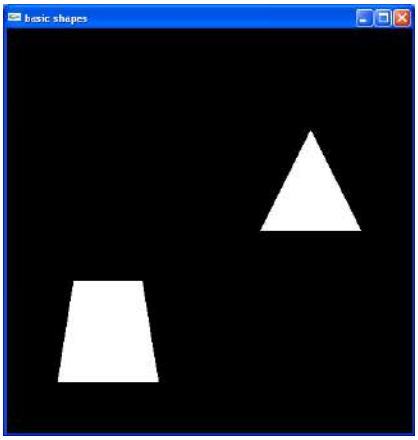


basicshapes.c

```
/* basicshapes.c  This program draws a white rectangle on a black
background. */

#include <GL/glut.h>          /* glut.h includes gl.h and glu.h*/

void drawScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

       //Triangle
       glBegin(GL_TRIANGLES); //Begin triangle coordinates
```

```
            glVertex2f(0.25, 0);
            glVertex2f(0.75, 0);
            glVertex2f(0.5, 0.5);
      glEnd(); //End triangle coordinates

      //Trapezoid
      glBegin(GL_QUADS); //Begin quadrilateral coordinates
            glVertex2f(-0.75, -0.75);
            glVertex2f(-0.25, -0.75);
            glVertex2f(-0.33, -0.25);
            glVertex2f(-0.67, -0.25);
      glEnd(); //End quadrilateral coordinates

       glFlush();
}

int main(int argc, char** argv)
{
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(0,0);
      glutCreateWindow("basic shapes");
      glutDisplayFunc(drawScene);
      glutMainLoop();
}
```




basic shapes